AD-A037 971 MICHIGAN UNIV ANN ARBOR SYSTEMS ENGINEERING LAB F/G 12/1
VECTORIZED GENERAL SPARSITY ALGORITHMS WITH BACKING STORE.(U)
JAN 77 D A CALAHAN, P G BUNING, W N JOY AF-AFOSR-2812-75
UNCLASSIFIED SEL-96 AFOSR-TR-77-0259 NL

1 OF 2
AD A037971

SEL Report # 96

Vectorized General Sparsity Algorithms with Backing Store

D. A. Calahan
P.G. Buning
W. N. Joy

January 15, 1977

Sponsored by Directorate of
Mathematical and Information Sciences,
Air Force Office of Scientific Research
under Grant 75-2812

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
SYSTEMS ENGINEERING LABORATORY
THE UNIVERSITY OF MICHIGAN, ANN ARBOR

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-77-0259	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): VECTORIZED GENERAL SPARSITY ALGORITHMS WITH BACKING STORE,	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): D. A. Calahan, P. G. Buning, W. N. Joy	8. CONTRACT OR GRANT NUMBER(s): AF-AFOSR 75-2812-75
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Dept. of Elec. and Comp. Engineering, University of Michigan, Ann Arbor, Michigan 48109	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 2304/A2, 61102F
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research (NM), Bolling AFB, Washington D. C. 20332	12. REPORT DATE: 15 January 1977
	13. NUMBER OF PAGES: 106
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): SEL-96	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Sparse matrices
Linear algebra
Parallel processing
Vector processing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The direct solution of large, sparse unsymmetric sets of simultaneous equations is commonly involved in the numerical solution of algebraic, differential, and partial differential equations. This report describes two new classes of computational algorithms for the solution of such equations. Each algorithm detects matrix structure suitable for vector processing and, potentially, for faster processing on cache machines. One procedure favors struc-

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 400 704

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ture usually associated with small sparse matrices; one is directed toward sets of equations requiring a large backing store. Comparisons of timing (on a cache machine) and of memory requirements are made between these new procedures and existing general sparsity techniques for a variety of science-engineering examples. Issues related to implementation are given for software implementations of the two algorithms.

VECTORIZED GENERAL SPARSITY ALGORITHMS WITH BACKING STORE

by

D.A. Calahan
P.G. Buning
W.N. Joy

SEL Report #96
January 15, 1977

Systems Engineering Laboratory
University of Michigan
Ann Arbor, Michigan 48109

Sponsored by Directorate of
Mathematical and Information Sciences
Air Force Office of Scientific Research
under Grant AFOSR 75-2812

400 704

Abstract

The direct solution of large, sparse unsymmetric sets of simultaneous equations is commonly involved in the numerical solution of algebraic, differential, and partial differential equations. This report describes two new classes of computational algorithms for the solution of such equations. Each algorithm detects matrix structure suitable for vector processing and, potentially, for faster processing on cache machines. One procedure favors structure usually associated with small sparse matrices; one is directed toward sets of equations requiring a large backing store. Comparisons of timing (on a cache machine) and of memory requirements are made between these new procedures and existing general sparsity techniques for a variety of science-engineering examples. Issues related to implementation are discussed. Finally, flowcharts and other user information are given for software implementations of the two algorithms.

TABLE OF CONTENTS

	page
Preface	i
Chapter 1. SPARSE MATRIX METHODS	1
A. Introduction	1
1. Historical review of direct sparse matrix algorithms	1
2. General vs. special sparsity algorithms	2
3. Summary of report	3
B. Sparse Matrix Factorization	4
1. Matrix factorization	4
2. Software implementation	5
C. Vectorization	7
1. Vectors and vector instructions	7
2. Vectorization efficiency	9
Chapter 2. VECTORIZED GENERAL SPARSITY ALGORITHMS	11
A. Row vs. Column Ordering	11
B. The LU Map Approach and Its Symbolic Vectorization	14
1. Introduction	14
2. Vectorized list data structures	18
3. Vector fills	19
C. Comparison of Scalar and Vector Sparsity Methods	20
1. Introduction	20
2. Storage	20
3. Speed: symbolic	21
4. Speed: numeric	21
D. Symbolic vs. Numeric Speed	25
E. Inner Loop Considerations	27
1. Introduction	27
2. Expanded vs. packed list structures	28
3. Sequencing of the multiply-subtract operations	30
4. Assembly-language programming	31
F. Partitioning	32
1. Introduction	32
2. Fixed local store	33
a. Introduction	33
b. Full matrices	34
c. Sparse matrices	35
d. The I/O problem for finite element solutions	39
e. The I/O problem in the substitution process	41
3. Variable local store	44
G. Evaluation	47
Chapter 3. DESCRIPTION OF A DUAL VECTORIZED GENERAL SPARSE EQUATION SOLVING PACKAGE	50
A. Introduction	50
B. General Program Description and Use	53
1. Program description	53
2. Example use of VEGES	53

C. Simplified Equation Formulation 58
1. Introduction 58
2. Scalar case: example 58
3. Scalar case: algorithm 61
4. Vector case: introduction 62
Unpartitioned form 63
Partitioned form 68
5. Vector case: algorithm 69
D. Program Flow Charts 70
E. Details of the Partitioned Solution 72
1. Introduction 72
2. Flow chart of UNBLOK symbolic preprocessor 73
3. Flow chart of ABLOK numeric preprocessor 73
4. Flow chart of symbolic, numeric solution 77
5. Example of partitioned solution of finite element problem 78
F. Fortran Implementation (VEGES) 83
1. Symbolic processing 83
2. Numeric factorization 83
3. Forward and back substitution 85
4. Conclusions 86
G. Fortran Implementation (VEGES/P) 86
1. Symbolic processing 86
2. Partitioning 87
3. Strip numeric factorization 88
4. Numeric factorization 89
5. Forward and back substitution 90

APPENDIX A. NUMERICAL EXPERIMENTS 94

APPENDIX B. EXAMPLE FORTRAN CODES FOR FORWARD SUBSTITUTION 103

APPENDIX C. MAIN PROGRAM FOR SOLUTION OF FINITE ELEMENT GRID OF FIGURE 19 104

REFERENCES 106

Preface

This report considers a number of issues related to the vectorization and partitioning of general sparse matrix solution algorithms that have not previously appeared in the sparse matrix literature. Typically, these involve processor modeling and algorithm analysis, design, and evaluation, particularly as they relate to the implementation and use of a software package prepared by the authors. With this rather broad topical coverage - perhaps suitable for several reports - it is felt to be useful to present the major results with page references, for the guidance of readers with specific research or software interests.

Vectorization

1. The sparse solution is decomposed into symbolic and numeric phases; in contrast to previously proposed scalar algorithms, this vectorized version results in relatively less symbolic processing time for large matrices (page 21).

2. The average length (L_{ave}) of vectors processed in the inner loop of the numeric phase, together with the nature of the processor (scalar or vector), determine the relative efficiency of the vector algorithm. This vector algorithm may be preferred even on scalar processors if L_{ave} is sufficiently large (pages 21-25).

3. Timing comparisons with a variety of other sparse solvers are given for a family of finite element problems (Appendix A).

Partitioning

1. I/0 transfer time to a backing store is an important issue, especially for the forward and back substitution steps; however, careful accounting in the symbolic processing phase can be used to predict when computation is I/0 bound or when the cost of local store in a virtual system becomes prohibitive (pages 33-38).

2. The megaflop rate, popular in evaluating vector processors, can be used to succinctly display the efficiency of partitioned general sparsity algorithms vis-a-vis special full-, band- and block-solvers (pages 47-49)

Implementation

1. A partitioning of the numeric but not the symbolic phase is proposed; pros and cons of this choice are discussed (pages 72-73).

2. Aids to equation formulation, although strictly not required in sparse equation software, are proposed to avoid cumbersome data structure and structural ordering and overlap problems (pages 58-69).

3. An interactive partitioning scheme, based on user specification of either column breaks or maximum buffer storage, is proposed (pages 77-82).

4. Examples - including Fortran code - of use of a software package on partitioned and unpartitioned problems are given (pages 56,104-105).

5. Timing results of applying two new sparse solvers to problems in electrical power systems, rigid body dynamics, and electronic devices, and to a family of finite element problems are presented (Appendix A).

CHAPTER 1. SPARSE MATRIX METHODS

A. Introduction

1. Historical review of direct sparse matrix algorithms

Prior to 1972, sparse systems of simulataneous equations tended to result from the solution of one of the following classes of equations:

(a) partial differential equation (PDE's), where, after discretization of the spatial variables, using finite difference (FD) or finite element (FE) methods, a regularly-structured matrix was solved using iterative techniques.

(b) ordinary differential equations (ODE's) or algebraic equations, where, after time discretization and/or linearization, an irregularly structured matrix was solved using direct methods.

In 1972, George [1] showed that, using dissection, a square n by n grid obtained from solution of PDE's by 5-point discretization formulae could be solved directly in $0(n^3)$ time rather than $0(n^4)$, a prohibitive cost associated with band-solution methods. Later, Woo and Gustavson [2] derived an ordering of the grid points which made dissection faster than band methods for n greater than 10. Considering that iterative methods require $0(n^2)$ time per iteration, these results made it feasible for the first time to solve many "small" PDE problems directly, leaving iterative methods only for large problems (perhaps n greater than 50) having special structural and numerical properties. Indeed, in [2] a variety of iterative and direct solution methods are compared to determine the value of n for which equal amounts of "work" are required. Very recently, Bank [3] showed that certain classes of FD problems can be solved

in $0(n^2)$ time.

2. General vs. special sparsity algorithms

When the matrix structure is highly regular and the number of spatial discretizations in the shortest dimension is small ($\cong$ 15), easily constructed band elimintation algorithms may be fully justified computationally. However,

(a) as the grid size grows in a FD or FE problem a dissection strategy is called for;

(b) as structural irregularity is introduced by curved or odd-shaped boundaries, by the use of a <u>family</u> of finite elements, or by implicit boundary conditions resulting from algebraic or ordinary differential equations of a physically-connected external system, a band algorithm becomes progressively less efficient; examples of the latter mixed PDE/ODE/algebraic systems are shown in Table 1;

Partial Dif. Eqns. (regular sparsity)	Ordinary Dif. and Algebraic Eqns. (irregular sparsity)
Structures	Mechanisms
Semiconductor Devices	Circuits
Combustion	Emissions

Table 1. Related applications yielding mixed sparsity structures

(c) when a vector computer architecture or a backing store is necessitated by a large problem size, specialized programming techniques may by required to achieve acceptable solution efficiency.

Dissection-related software to implement (a) exists in a variety of forms. Early programs required grid sizes related to a power of 2 and involved data structures that require recoding of the equation formulation step in a band-related solution. This complication induced George in a later publication [4] to consider less efficient dissection methods that retain some of the simplicity of band matrix programming.

General sparsity methods - where an arbitrary matrix structure is allowed - offer an alternative to both banded and dissected grid solution methods. General sparsity software can accept a matrix formulation in any order, since the equations can be reordered internally by the sparsity software to correspond to a banded or a dissected solution. Usually, the principal price extracted of the user for this generality is a preprocessing step where the matrix structure is analyzed and an efficient numerical solution algorithm prepared specfically for that structure. The introduction of additional structural irregularity for any of the reasons cited in (b) above is then at least conceptually trivial.

3. Summary of report

The above rationale for use of general sparsity software would apply to any of a number of existing software packages

[5], [6]. The aim of this report is to expand the role of such algorithms by

(a) proposing alternate data structures that improve the storage efficiency of existing software when solving large systems of equations associated with large FD and FE problems;

(b) including the optional use of a backing store when, because of problem size, either real memory capacity of a dedicated system is exceeded or total memory costs become a significant factor in a virtual environment:

(c) recognizing local structure in the matrix that can be exploited by a vector processor and, incidentally, can improve solution times on cache "scalar" processor as well.

Two increasingly sophisticated software pacakages are described which permit the reader to evaluate the programming effort involved in utilizing general sparsity techniques. As the problem size grows, backing store and special architectures can be brought into use with no change in the equation formulation and a minimum of additional programming effort.

B. Sparse Matrix Factorization

1. Matrix factorization

Consider the equation

$$\underline{A}\ \underline{x} = \underline{b}$$

where $\underline{A} = [a_{ij}]_{nxn}$, and $\underline{b} = [b_{ij}]_{nx1}$ contain constant coefficients

and $\underline{x} = [x_{ij}]_{nx1}$ is a vector of unknowns. We will choose to factor $\underline{A}$ into the triangular form

$$\underline{A} = \underline{L}\ \underline{U}$$

where $\underline{L} = [\ell_{ij}]_{nxn}$, a lower triangular matrix, and $U = [u_{ij}]_{nxn}$, an upper triangular matrix, and either $\ell_{ii} = 1$ for $1 \leq i \leq n$ or $u_{ii} = 1$ for $1 \leq i \leq n$. In general, $\underline{L}$ and $\underline{U}$ will contain the same non-zero positions as $\underline{A}$, plus "fill" positions created by the elimination process.

2. Software implementation

The use of specialized algorithms and data-handling methods for sparse equations is almost a decade old [7]. The random structure is usually described by a bit map or a linked list. This structure is then often preprocessed symbolically to reduce the computation in a subsequent (repeated) numerical solution phase (Figure 1). The three common two-step (symbolic/numeric) solution methods are:

(a) code generation [8], [6], where a large set of explicit machine instructions - including array indices - are generated in the symbolic step; these instructions "map" the given $\underline{A}$ and $\underline{b}$ into $\underline{x}$ during the numerical solution step; an instruction must be generated for each arithmetic operation, so that if a full matrix were being processed $0(n^3)$ instructions would be required corresponding to the $0(n^3)$ arithmetic operations;

(b) interpretive index generation [9], [6] similar to (a) except that the code is in the form of array indices and higher level instructions; for example, the instructions might specify row/column operations such as inner and outer product and

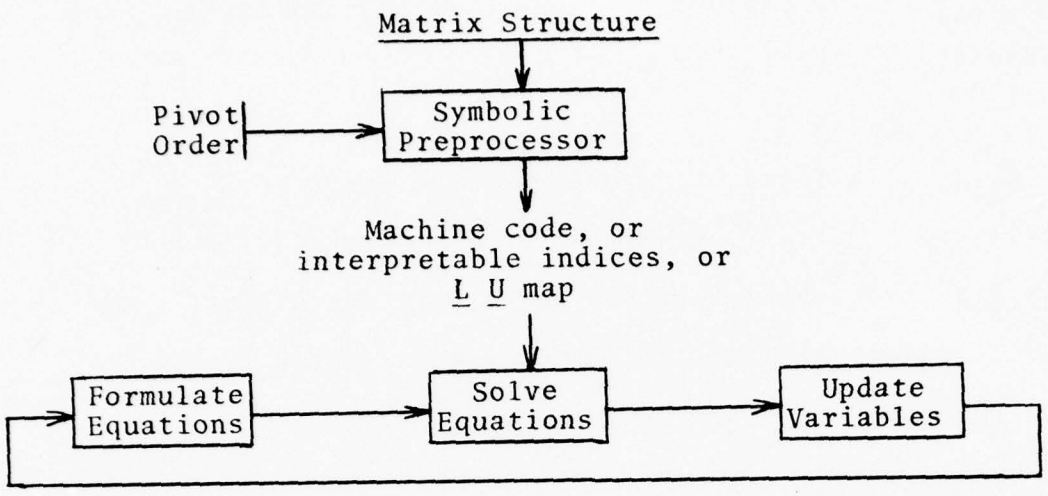


Figure 1. Model symbolic and numerical solution procedure

the indices would specify the non-zero positions of the row/column; less storage is required than (a) - usually a factor of 5 or 10 - but execution is commonly 3-5 times slower;

(c) LU map approach [10][12] where the map of $\underline{L}$ and $\underline{U}$ (i.e., $\underline{A}$ and its fills) is determined by the preprocessor; for a full matrix, only $0(n^2)$ storage is required for the map of L and U, the same as for the numerical values of the matrix itself; this procedure appears to be as fast as the method of (b)(see Table A3) and for these reasons (b) is not often used.

Detailed speed and storage comparisons of (a) and (c) for two FD examples are given in [2][11].

In this report, the LU map approach is chosen because of its reduced storage requirements. The storage map is further reduced by compressing the data structure for adjacent non-zero matrix positions, i.e., dense segments of rows/columns. Thus, the storage of the structure of a full matrix would be 0(n), since each row/column could be specified by the address of the first row/column position and by the row/column length (n).

It will be shown that recognition of such structure allows increased solution efficiency for cache "scalar" machines and can be expected to vastly improve the performance of vector processors. Besides preparing a map for the numerical solution, the symbolic preprocessor - far faster than the numerical solver for large matrices - is useful for predicting solution times and for optimizing the automatic or interactive partitioning of the solution when a backing store is used.

C. Vectorization

1. Vectors and vector instructions

Closely associated with the condensation of the storage map of L and U by recognition of higher level structure is the efficient processing of this structure by exploiting two features of modern scientific processors - pipelining and parallelism. The following discussion of the use of these features will be quite simplified; further descriptions and rationales will be presented in Chapter 2.

We will use the word vector to mean an array of data. Thus, a vector operation is an operation on arrays. This includes row/column operations on matrices, operations on submatrices (blocks), etc.

For purposes of later reference, we distinguish between

(a) a simple vector operation, to replace the Fortran-like operations

```
      DO 1 M = 1, N
1     C(I1(M)) = A(I2(M)) o B(I3(M))
```

where o indicates an arithmetic or logical operation and where $I_1(M)$, $I_2(M)$, and $I_3(M)$ are linear indexing functions of the form $I_i = \alpha_i + M\beta_i$ where α_i and β_i are arbitrary; thus, with o = +, $\alpha_i = 0$, $\beta_i = 1$, the arrays A and B are added to produce the array C. If o = *, $I_1 = M$, $I_2 = 1$, and $I_3 = M$, a "broadcast" multiply (multiplication of a vector by a scalar) results.

(b) a higher level vector operation to replace the triple (or double) loop

```
      DO 1 J = 1, N1
      DO 1 K = 1, N2
      DO 1 L = 1, N3                                          (4)
1     C(I1(J),I2(K),I3(L)) = A(I4(J),I5(K),I6(L)) o B(I7(J),I8(K),
                                                        I9(L))
```

where $I_i(M) = \alpha_i + M\beta_i$.

Computationally, the important characteristic common to (a) and (b) is that only one vector "startup" is required in each case. This startup may include time to determine N1, N2, N3 in a convectional (scalar) machine and/or to fill the arithmetic pipeline in a vector processor. In contrast, the "operate" time includes those operations that must be repeated for each pass through the (lowest level) loop, and will include floating point arithmetic, in addition to array indexing and loop termination test, depending on the machine architecture. In general, fewer vector startups result in less time devoted to overhead calculations and thus a higher overall efficiency.

2. Vectorization efficiency

This efficiency may be quantified as follows. The computation time to perform the i^{th} vector operation is

$$T_i = T_{s_i} + r_i T_{op_i} \tag{5}$$

where T_{s_i} is the startup time, T_{op_i} the operate time, and r_i the vector length. If T_{s_i} and T_{op_i} are independent of i, then the time to perform m vector operations is

$$T = mT_s + T_{op} \sum_{i=1}^{m} r_i$$

Since the operate time T_{op} is the useful computation time, define the vectorization efficiency as

$$\eta = \frac{\text{operate time}}{\text{startup time + operate time}}$$

$$= \frac{(T_{op}/T_s)\left(\sum_{i=1}^{m} r_i/m\right)}{1 + (T_{op}/T_s)\left(\sum_{i=1}^{m} r_i/m\right)} \tag{6}$$

Although T_{op}/T_s *is a machine parameter*, the quantity $\sum_{i=1}^{m} r_i/m = L_{ave}$ is problem-dependent, and identifiable as the "average vector length." Note that an efficiency of .5 is achieved when the average vector length is equal to the ratio T_s/T_{op}.

Although (6) usually applies to only a single class of

vector instructions (since T_{op_i} is assumed constant) similar formulae can be derived from (5) when several classes of vector instructions are involved. In the factorization problem, we will show that the inner loop consists of a vector multiply and a vector subtract, each of the same length. It is easily demonstrated that the efficiency obtained from (5) is then the same as for a single instruction with startup and operate times equal to the sum of startup and operate times for a subtract and multiply. The average vector length of the single time-equivalent instruction is therefore a useful concept.

CHAPTER 2. VECTORIZED GENERAL SPARSITY ALGORITHMS

A. Row vs. Column Ordering

For purposes of this discussion, assume that the equations represented by $\underline{A}\ \underline{x} = \underline{b}$ cannot be locally decoupled, so that only single rows/columns may be pivoted upon at a time (see [13] for multi-row elimination). Associated with an n x n matrix, n pivot steps can be identified, each (r^{th}) step involving a division of the r^{th} row or column by the pivot element and a sequence of multiply-subtract operations involving the r^{th} row or column and other rows or columns of the matrix.

Although we will have reason later to study a variety of factorization algorithms that involve backing store and architectural issues, for the present two rather conventional procedures will be compared.

a). Row-ordering

Let

$$(\underline{u}^{(k)}_{r+1,n})^T = [u^{(k)}_{r,r+1} \cdots u^{(k)}_{r,n}] \tag{7}$$

$$(\underline{\ell}^{(k)}_{j,n})^T = [\ell^{(k)}_{r,j} \quad \ell^{(k)}_{r,j+1} \cdots \ell^{(k)}_{r,r}] \tag{8}$$

where the subscripts on the vectors correspond to beginning and ending column numbers, and where $u^{(0)}_{r,m} = a_{r,m}$, $\ell^{(0)}_{r,m} = a_{r,m}$. The row-wise factorization step is described by

$$[(\ell^{(k)}_{k+1,r})^T \quad (\underline{u}^{(k)}_{r+1,n})^T] = [(\underline{\ell}^{(k-1)}_{k+1,r})^T \quad (\underline{u}^{(k-1)}_{r+1,n})^T] - \underline{\ell}_{r,k}\ \underline{u}_{k+1,n} \tag{9}$$

$$k = 1,2,\ldots r-1$$

$$\underline{u}_{r+1,n} = \frac{1}{\ell_{rr}} \underline{u}^{(r-1)}_{r+1,n}$$

$$\underline{\ell}_{1,r} = \underline{\ell}^{(r-1)}_{1,r}$$

The forward and back substitution steps require the solution of $\underline{L}\,\underline{y} = \underline{b}$, $\underline{U}\,\underline{x} = \underline{y}$. Using the same vector notation as (7-8)

$$y_1 = b_1$$
$$y_r = (b_r - (\underline{\ell}_{1,r-1})^T \cdot \underline{y}_{1,r-1})/\ell_{rr} \qquad r = 1,2,\ldots n \quad (10)$$

$$x_n = y_n$$
$$x_r = y_r - (\underline{u}_{r+1,n})^T \cdot \underline{x}_{r+1,n} \qquad r = n-1,n-2,\ldots 1 \quad (11)$$

The row ordering of $\underline{L}$ and $\underline{U}$ requires the inner product of two vectors.

b). Column-ordering

Let

$$\underline{u}^{(k)}_{j,r} = \begin{bmatrix} u^{(k)}_{j,r} \\ \cdot \\ \cdot \\ \cdot \\ u^{(k)}_{r,r} \end{bmatrix} \qquad \underline{\ell}^{(k)}_{r+1,n} = \begin{bmatrix} \ell^{(k)}_{r+1,r} \\ \cdot \\ \cdot \\ \cdot \\ \ell^{(k)}_{n,r} \end{bmatrix}$$

where $u^{(0)}_{m,r} = a_{m,r}$, $\ell^{(0)}_{m,r} = a_{m,r}$. The columnwise factorization is

$$\begin{bmatrix} \underline{u}^{(k)}_{k+1,r} \\ ----- \\ \underline{\ell}^{(k)}_{r+1,n} \end{bmatrix} = \begin{bmatrix} \underline{u}^{(k-1)}_{k+1,r} \\ ----- \\ \underline{\ell}^{(k-1)}_{r+1,n} \end{bmatrix} - u_{k,r} \; \underline{\ell}_{k+1,n} \qquad k = 1,2,\ldots r-1 \quad (12)$$

$$\underline{\ell}_{r+1,n} = \frac{1}{u_{rr}} \underline{u}^{(r-1)}_{r+1,n}$$

$$\underline{u}_{1,r} = \underline{u}^{(r-1)}_{1,r}$$

and the forward and back substitution step is

$$\begin{aligned} \underline{y}^{(0)}_{1,n} &= \underline{b} \\ \underline{y}^{(r)}_{r+1,n} &= \underline{y}^{(r-1)}_{r+1,n} - y_r \, \underline{\ell}_{r+1,n} \qquad r = 1,2,\ldots n-1 \end{aligned} \tag{13}$$

$$\left.\begin{aligned} \underline{x}^{(n)}_{1,n} &= \underline{y}^{(n-1)}_{1,n} \\ x_n &= x^{(n)}_n / u_{nn} \\ \underline{x}^{(r)}_{1,r} &= \underline{x}^{(r+1)}_{1,r} - x_r \, \underline{u}_{1,r-1} \\ x_r &= x^{(r)}_r / u_{rr} \end{aligned}\right\} \quad r = n-1,\ldots 1 \tag{14}$$

The choice of row- or column-ordered algorithm can be significant for a scalar processor (to be shown experimentally later). For a vector processor having an efficiently-implemented inner-product instruction, row ordering is preferred; however, a processor with chained multiply-add arithmetic units such as the Cray-1 clearly suggests use of column-ordering. The reader will recognize, however, that only the forward and back substitutions need be changed to accommodate row-ordering since $u_{rr} = 1$ and $\ell_{rr} = 1$ in the row- and column-ordered methods, respectively. These options are available in subroutines (VMBPR, ZMBPR) and (VMBPC, ZMBPC) respectively in the software package of Chapter 3.

B. The LU Map Approach and Its Symbolic Vectorization

1. Introduction

The purpose of the symbolic phase of Figure 1 is to determine the fill characteristics of A, i.e., the exact structure of L and U. This information is used by the numeric part to reduce the solution time (Gustavson [10] cites a factor of 2-3 for the LU map approach).

To acquaint the reader with this approach an example using Gustavson's "scalar" map is shown in Table 2. Special note should be taken of

(1) the fill positions detected by the symbolic phase in the generation of the LU map;

(2) the use of map indices in the numeric solution to extract information from the numeric arrays A, L, and U;

(3) the use of an expanded current column (X array), requiring zeroing, expansion, and contraction in the loading and storing process (see [10] and page 28 of this report for alternative procedures);

(4) the opportunities for the use of (simple) vector operations in the numeric solution, as evidenced by the indexed array operations marked "vector".

It should be pointed out that fill detection is essential to any sparse matrix factorization algorithm. The LU map approach exploits the fact that this costly process need be performed only once for a given matrix structure, allowing multiple solutions with different numerical values - as occurs in a Newton linearization process.

$$A = \begin{matrix} 3 & 0 & 0 & 0 & 2 \\ 0 & 4 & 2 & 1 & 0 \\ 0 & 2 & 6 & 0 & 3 \\ 0 & 1 & 0 & 3 & 1 \\ 2 & 0 & 3 & 1 & 5 \end{matrix} \qquad LU = \begin{matrix} 3 & 0 & 0 & 0 & 2 \\ 0 & 4 & 2 & 1 & 0 \\ 0 & 1/2 & 5 & -1/2 & 3 \\ 0 & 1/4 & -1/10 & 27/10 & 13/10 \\ 2/3 & 0 & 3/5 & 13/27 & 67/54 \end{matrix}$$

current column

matrix — completely-factored matrix

from user

A (column-ordered numeric values of A matrix)
3,2,4,2,1,2,6,3,1,3,1,2,3,1,5

JA (JA(j) points to beginning of jth column of A in IA)
1,3,6,[9],12,16

IA (column-ordered list of row numbers of A)
1,5,2,3,4,[2,3,5],2,4,5,1,3,4,5

generated by symbolic

JL (JL(j) points to beginning of jth column of L in IL)
1,[2,4,6],7

IL (column-ordered list of row numbers of L)
5,[3,4,4,5],5 fill

JU (JU(j) points to beginning of jth column of U in IU)
1,1,1,2,4,7

IU (column-ordered list of row numbers of U)
2,[2,3],1,3,4 fill

generated by numeric factorization

L (column-ordered numeric values of L)
2/3,1/2,1/4,-1/10,3/5,

U (column-ordered numeric values of U)
2,_,_,_,_,_

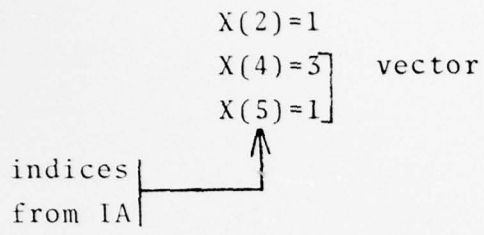
DI (ordered numeric values of diagonal)
3,4,5,_,_

(a) Example up to factorization of fourth column

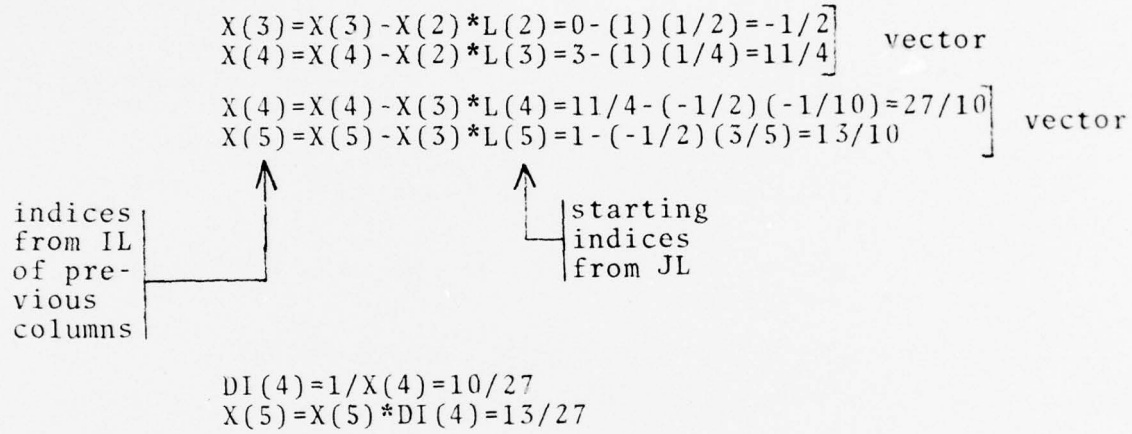
Table 2. Example of use of LU map in factorization

1. Zero expanded current column (X array)

2. Load current column with fourth column of A

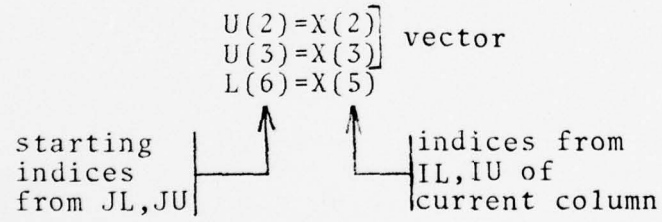


3. Factorize fourth column



DI(4)=1/X(4)=10/27
X(5)=X(5)*DI(4)=13/27

4. Store current column



(b) Steps in Factorization of Fourth Column

Table 2. Example of use of LU map in factorization

The details of the symbolic map generation are left for Chapter 3. However, the following two sections are intended to give insight into this process by discussion of vectorized data structure and symbolic operations on it during the factorization process.

2. Vectorized list data structures

Consider a column of a sparse matrix having the non-zero row positions shown in Figure 2 (before fill). This structure would be described in a conventional ordered list as

31,32,...36,39,42,43,...47 (15)

Such a list enumerating all row positions will be termed scalar storage. Clearly, the list can be shortened by identifying sets of contiguous positions (vectors) and retaining only the first and last row numbers, viz,

31,36,39,39,42,47 (16)

This form is natural to looping operations for a scalar processor, where pairs of numbers are directly usable as upper and lower loop indices. Alternatively, the initial row position and the vector length could be stored as

31,6,39,1,42,6 (17)

This form is favored by vector processors with hardware that counts down vector arithmetic operations to terminate a vector operation.

Another choice, preferred when a significant number of singleton (scalar) positions are present, represents a vector of length one with a minus sign prefixing the row number as

31,36,-39,42,47 (18)

This latter structure has been adopted in this report.

3. Vector fills

The multiply-subtract operation of (12) can result in production of fills that must be detected in the symbolic phase. In Figure 2, the process of multiplying the k^{th} column of $\underline{L}$ (termed a preceeding or recalled column) by $u_{k,r}$ and subtracting from the

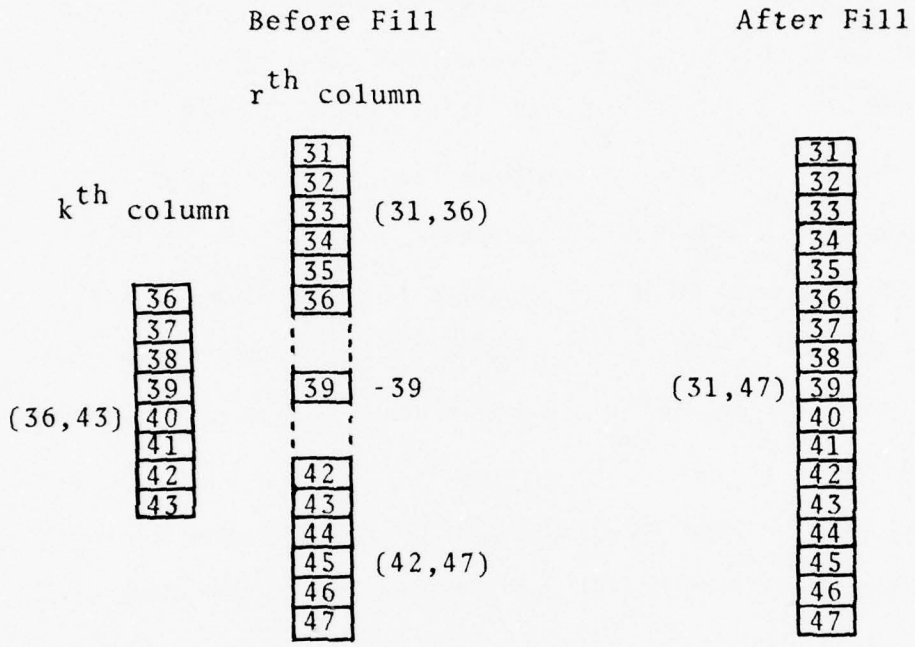


Figure 2. Example of vector fill, with data structure description (scalar indicated by - sign)

r^{th} column of $\underline{L}$ (termed the current column) and $\underline{U}$ is depicted. The zero-valued positions 37,38,40,41, which initially separate two vectors and a scalar, are filled by the dense vector (36,43) in the k^{th} column.

The symbolic phase produces the *LU map* by scanning the number pairs representing the vector structure of all the preceeding columns and the current column to determine zero-valued regions of

the latter covered by at least one of the former. These are the fill positions.

C. Comparison of Scalar and Vector Sparsity Methods

1. Introduction

Although one could expect to reduce storage by compacting vectors in the data structure and could hope to exploit the structure of machines architected to efficiently process looped instructions, it is less obvious that conventional (scalar) machine performance would benefit from vectorization. In this section we study both the storage and the speed issues in some detail, showing quantitatively the advantage of vectorization even for one of the most recent scalar processors.

2. Storage

In Gustavson's LU map approach, the column ordered map of $\underline{L}$ and $\underline{U}$ is saved in arrays IL and IU. Thus, for every numerical value in $\underline{L}$ and $\underline{U}$, there is a symbolic value in IL or IU (Table 2).

Now consider an LU map with m vectors of average length ℓ. The scalar and vector maps require $m\ell$ and 2m locations respectively, exclusive of singletons. Now consider the symbolic (2-byte) and numeric (8-byte) storage. Define

$$\text{storage factor} = \frac{\text{vectorized storage}}{\text{scalar storage}}$$

which becomes

$$\text{storage factor} = \frac{2(2m) + 8(m\ell)}{2(m\ell) + 8(m\ell)}$$

$$= .8 + \frac{.4}{\ell}$$

Thus, a vectorized map will result in a 20% storage savings as $\ell \to \infty$.

3. Speed: symbolic

A speed improvement is possible in both the symbolic and numeric processing stages.

Consider again in Figure 2 the symbolic process of creating a vector fill. Since each vector is described by a pair (beginning row position, ending row position), the length of the data structure does not depend on the vector lengths. Similarly, the operations performed on this structure to determine fill are not changed if, for example, all vector lengths are doubled. Obviously, this is not true for the scalar approach, where each row position must be examined separately for fill.

In Table A3 of the appendix, symbolic solution times are given for both scalar and vector versions, each processing a family of finite element matrices. For matrices of dimension 9 and 49, the scalar version is faster; the vector version is nearly three times the speed of the scalar for a matrix of dimension 961.

4. Speed: numeric

A more important comparison of the scalar and vector approaches involves the repeated numerical processing stage. Although computer architecture characteristics - particularly the extent of instruction and operand pipelining - are quite influential here, nonetheless it is possible to construct a simplified model of a typical vectorizeable operation and discover the conditions in which the scalar approach may be preferable.

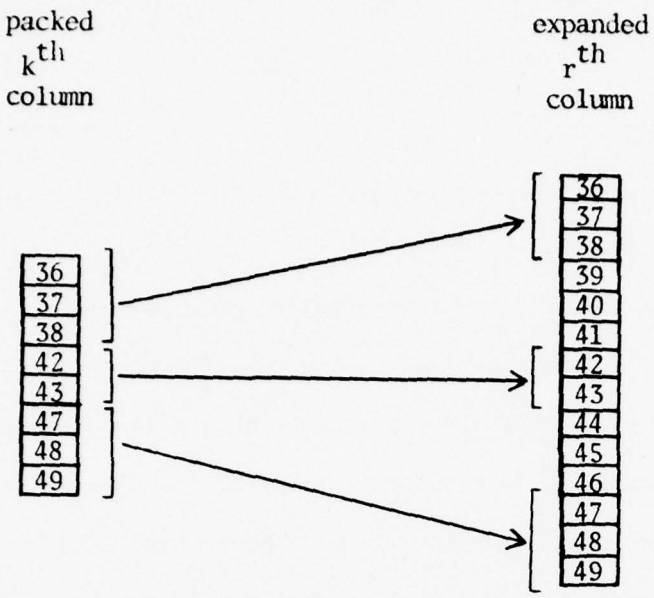


Figure 3. Subtraction of packed (k^{th}) column from expanded (r^{th}) current column

The most time-consuming part of the factorization algorithm is the multiplication and subtraction step of (12). The numerical values of the k^{th} column are assumed packed as illustrated in Figure 3. Multiplication by $u_{k,r}$ can therefore be performed in a loop or vector mode without reference to the row positions. However, the subtraction of (12) involves knowledge of the row positions; this is shown in Figure 3, where three segments of the k^{th} column are depicted being subtracted from the expanded r^{th} column. This in turn requires that the row position must be addressed either one at a time - one for each numerical value - or in a vector sense - the beginning and ending row address of a dense column segment. A Fortran implementation of this subtraction process for a column in the numeric phase would be of the form

```
      DO 2 J=N1,N2
      K=IL(J)
2     X(K)=X(K) - Y(J+L)
```

Scalar

```
      N3=IL(N1)
      N4=IL(N1+1)
      DO 2 J=N3,N4
2     X(J)=X(J) - Y(J+L)
```

Vector

Here, the current column (X array) is addressed through the IL array in the scalar case; in the vector case, IL contains the beginning and ending row numbers of a dense segment of the k^{th} column.

Clearly, the vector algorithm requires more startup time - to compute N3 and N4 - but less loop execution time than the scalar case, which involves indirect addressing.

It happens that these simplified loops are not representative of either the factorization or the substitution steps. A more extended test is given by the loop timing program of Table 3, where vectorized and scalar versions of both inner product and broadcast multiply-subtract implementations of the inner loop are timed. The inner product is usually associated with a row-ordered substitution process (see Section A) whereas the broadcast operation is used in row-ordered factorization, column-ordered factorization, and column-ordered substitution. All loop operations may be characterized by a loop startup (T_s) and a loop execution (T_{op}) in the manner of Eq. (5); experimental results are given in Table 4 in terms of these parameters.

Loop description	Startup (μsec)	Execution (μsec)
Vector inner product	2.6	1.6
Scalar inner product	1.	1.7
Vector broadcast-subtract	2.8	2.24
Scalar broadcast-subtract	.72	2.42

Table 4. Loop timing results; Amdahl 470V/6,Fortran H, double precision

Comparing the two broadcast loops, the vectorized version has a

```
C****  TIMING TEST FOR INNER LOOPS OF ALTERNATE FACTORIZATION
C       AND SUBSTITUTION ALGORITHMS
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION X(26000),Y(26000),JC(50000)
      DO 12 J=1,26000
      X(J)=1.D-50
12    Y(J)=X(J)
 6    DO 1 J=1,5000
1     JC(J)=J
      XA=1.
      READ(5,22)LVECT
      NTIMES=50000/LVECT
      WRITE(6,22)LVECT,NTIMES
22    FORMAT(2I5)
      L=5
      N1=1
      N2=LVECT
C****  TEST FOR GUSTAVSON FACTORIZATION (SCALAR)
      CALL TIME(0)
      DO 3 M=1,NTIMES
      DO 2 J=N1,N2
      K=JC(J)
2     X(K)=X(K)-XA*Y(J+L)
3     CONTINUE
      CALL TIME(1,1)
C****  TEST FOR GUSTAVSON F. & B. SUB. (SCALAR)
      SUM=0.D0
      CALL TIME(0)
      DO 7 M=1,NTIMES
      DO 8 J=N1,N2
      K=JC(J)
8     SUM=SUM-X(J)*Y(K)
7     CONTINUE
      CALL TIME(1,1)
      N1=0
      DO 11 J=1,50000,2
      JC(J)=URAND(INIT)*1000+1
11    JC(J+1)=LVECT+JC(J)-1
      N5=100000
C****  TEST FOR VECTORIZED FACT. AND SUBSTITUTION
      CALL TIME(0)
      DO 5 M=1,NTIMES
      IVTI=0
9     N1=N1+1
      N3=JC(N1)
      N1=N1+1
      N4=JC(N1)
      JDIFF=IVTI-N3+1
      DO 4 J=N3,N4
4     X(J)=X(J)-XA*Y(J+JDIFF)
      IVTI=JDIFF+N4
      IF(N1.GT.N5)GO TO 9
5     CONTINUE
      CALL TIME(1,1)
      N1=0
C****  TEST FOR VECTORIZED INNER PRODUCT
      SUM=0.D0
      CALL TIME(0)
      DO 15 M=1,NTIMES
      IVTI=0
10    N1=N1+1
      N3=JC(N1)
      N1=N1+1
      N4=JC(N1)
      JDIFF=IVTI-N3+1
      DO 14 J=N3,N4
14    SUM=SUM-XA*Y(J+JDIFF)
      IVTI=JDIFF+N4
      IF(N1.GT.N5)GO TO 9
15    CONTINUE
      CALL TIME(1,1)
      GO TO 6
      END
```

Table 3. Timing program for inner loops

larger startup but a smaller loop execution time. The total loop timings become equal when (ℓ is the loop length)

$$.72 + 2.42\ell = 2.8 + 2.24\ell$$

or $\ell = 11$. Thus, a scalar processor is found to operate preferentially on vectors; as $\ell \to \infty$ the loop execution time of the vectorized version is 2.24/2.42 = .92 the time of the scalar.

These timings will be used throughout the report to estimate the computation times for large problems. To illustrate this use, consider the problem of estimating the relative times of scalar and vector factorization of the 961 x 961 matrix of Table A3 using broadcast inner loops. The average vector length (L_{ave}) of this loop is taken from Table A2 as 6.85 elements. Therefore,

$$\frac{\text{scalar loop time}}{\text{vector loop time}} = \frac{.72 + 2.42(6.85)}{2.8 + 2.24(6.85)} = .95$$

This compares with a fraction for the factorization step from Table A3 of 1146/1309 = .88. Both ratios decrease for the smaller matrices in the finite element family studied, as L_{ave} becomes smaller and the vector loop startup becomes more significant.

D. Symbolic vs. Numeric Speed

As the average vector length increases, the symbolic processing time should decrease relative to the numeric time. To quantify this concept, consider a recalled column of $\underline{L}$ with m vectors of average length L_{sub} being multiplied by a scalar and subtracted form the current column. The symbolic processing can be expected to be proportional to m and the numeric proportional to mL_{sub}. Therefore,

$$\frac{\text{N(numeric processing time)}}{\text{S(symbolic processing time)}} = K\, L_{sub} \qquad (19)$$

This simplified analysis yields surprisingly consistent experimental results. The L_{sub} is the average vector length of the subtract operation in the inner loop, a quantity that can be measured experimentally and calculated precisely for the dissected finite element grid. In Table 5, the K of (19) is shown for all the sparse matrices studied. For the wide range of matrix size and structure,

$$.24 \leq K \leq .31.$$

Using K = .25, one can estimate from Table A3 the N/S ratio for the general finite element grid as

$$\frac{N}{S} = \frac{(.13)2^n}{n-2.96}$$

giving for a 126 x 126 grid N/S = 4. Since the symbolic phase involves mainly comparison operations with ordered pairs of numbers (to establish fill regions) it can not be expected to be itself vectorizeable. Thus the above value for K would be much smaller if both phases were run on a vector machine. The symbolic phase - viewed as a vectorization step - would most efficiently be executed on a scalar processor, and the numeric phase on a vector processor.

As the N/S ratio becomes larger, the symbolic step becomes useful as a simulation tool. The number, length, and type of vector operations can be determined in the symbolic and, knowing the functional characteristics of a processor, computation times can be predicted. This is especially useful for estimating solution times of vector processors that are not conveniently available. Such a simulation program has veen devised and has been useful in obtaining precise vector-related structural information such as displayed in Tables A2 and A5.

		N/S	L_{sub}	$K = (N/S)/L_{sub}$
1.	Dissected Rect. Grid Matrix (grid) size			
	9 x 9 (2 x 2)	.53	1.97	.27
	49 x 49 (6 x 6)	.69	2.92	.24
	225 x 225 (14 x 14)	1.12	4.42	.25
	961 x 961 (30 x 30)	1.7	6.85	.25
2.	Electric Power Problem	.676	2.6	.26
3.	Aircraft Landing System	.500	1.6	.31
4.	Electronic Device Model	4.76	16.3	.29

Table 5. Experimental determination of K in Equation 19, for an Amdahl 470 V/6, double precision, Fortran H.

E. Inner Loop Considerations

1. Introduction

For large matrices, the multiply-subtract inner loop becomes the dominant computation. To illustrate, inner loop operations for the factorization of the largest finite element matrix of Appendix A involves 59026 inner loop startups and 404587 loop executions. From the benchmark of Table 4, the inner loop time can be estimated as

$$T_{inner} = (59026)(2.8) + (404587)(2.24)$$
$$= 1.07 \times 10^6 \ \mu sec.$$

This is 81 percent of the measured factorization time of 1.309 sec. of Table A3. Since this estimate accounts for the time devoted to the multiplication of a scalar from $\underline{U}$ by a column from $\underline{L}$, the percentage of time devoted to the inner loop computation will increase with the column density of $\underline{L}$; from Table A1, this is a logarithmically-increasing function of grid size.

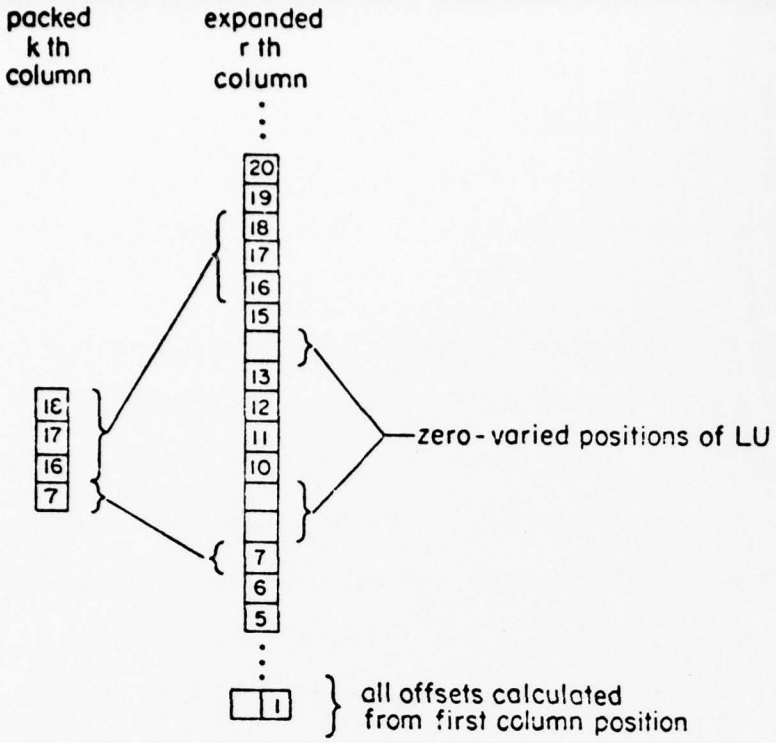
In the following sections, issues related to the implementation of this loop are discussed; the scalar-vector comparisons have been made in section C4.

2. Expanded vs. packed list structures

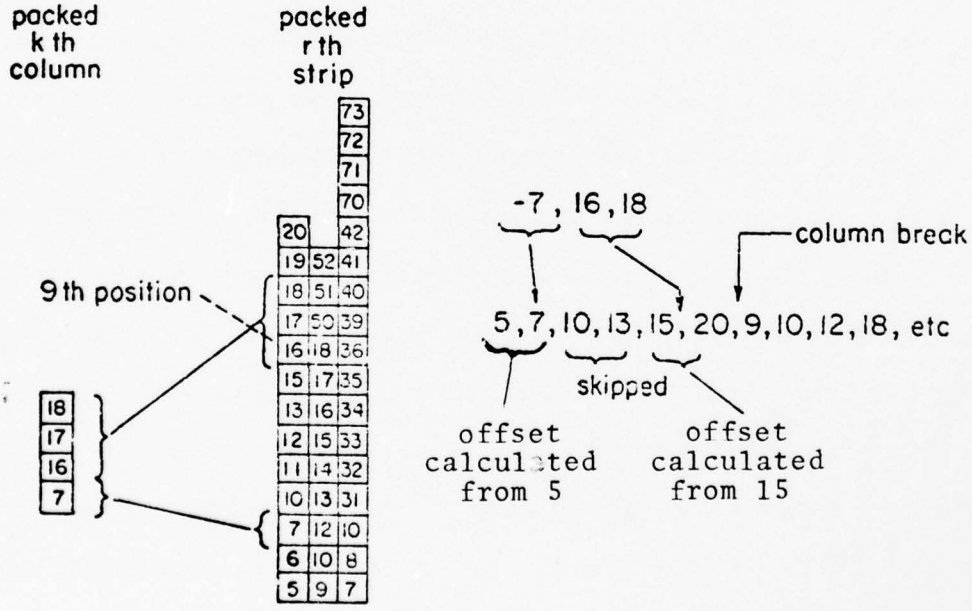
The inner loop of the factorization involves a multiply of a packed column of $\underline{L}$ by a scalar and a subtraction of the result from the current column being factored. The multiply can be carried out in one vector multiply; the subtraction must be performed with attention to row numbers of the minuend and subtrahend. This requirement suggests at least two procedures based on the data structure of the current column.

1) If the current column is fully expanded, subtraction can take place using the address of the initial row position of the current column as a reference; as illustrated in Figure 4a, a vector in the packed k^{th} column can be subtracted from the r^{th} column with only a single address offset calculation. The current column must be expanded from the initially packed matrix and must be compacted after the last subtraction, usually a trivial 0(n) process. Also, the storage demanded by this expansion process is usually not a problem even with widely-separated elements or blocks of elements.

2) If the current column is packed with an accompanying vectorized list description of the row numbers of $\underline{L}$ and $\underline{U}$, then subtraction involves searching the list until the row number(s) of the minuend are in the range of those of the subtrahend; in Figure 4b, for example, the current list would be scanned for subtraction of the scalar in row 7 and the vector in rows 16-18; the locations of the numerical values in the packed current vector would then be calculated for the vector in rows 16-18 as (for example)



(a) Expanded current column of LU



(b) Packed current column of LU

Figure 4. Illustration of subtraction operations on expanded, packed list structures.

$$[(7-5)+1] + [(13-10)+1] + [(16-15)+1] = 9$$

denoted on Figure 4b. Thus, a running total of the offset of each vector of the current column must be maintained as the list is scanned.

The scanning and address calculation is clearly overhead which must be small relative to the subtraction time itself. Thus, the number of vectors - represented by the list length - must be small and their size must be large. This is in fact a characteristic of matrices arising from large FD and FE problems, where each column will have on the average 4-6 vectors with an average length of 30 or more numerical values . Smaller FD or FE problems or matrices not having these characteristics are more efficiently solved with the expanded storage of (1). For these reasons, the programs to be later described were written using both procedures: the program using only local store for small matrices will have expanded column storage; the program allowing a backing store will use a packed current column throughout. The third data column of Table A3 and the seventh column of Table A4 yield a comparison of timings resulting from expanded and packed current column storage; e.g., for a 961 x 961 matrix, factorization requires 1309 msec and 1667 msec, respectively. The relative speeds as a function of n show a fractional difference decreasing toward 1 between the two timings, the above ratio being only 1.22. Undoubtedly, the increasing L_{ave} with problem size is primarily responsible for decreasing overhead in traversing the packed column lists.

3. Sequencing of the multiply-subtract operations

Another inner loop issue introduced by vectorized solution is whether the multiplication of a column of L will be performed in a

3. Variable local store

single vector operation or whether the multiplication and subtraction will be performed within the same loop. The latter has the advantage in a scalar processor that fewer loads and stores are necessitated, since the result of the multiplication is available either in a register or in cache memory for the following subtraction. However, fewer vector multiplies (i.e., startups) are required for the former process. Thus both options should be available in a general program. Note that this consideration appears in the forward and back substitution as well. The third data column of Table A3 shows the significant impact in a cache machine of combined multiply-subtract operations (the number in parenthesis is the timing with separate operations). For a 961 x 961 matrix, the ratio is 1773/1309 = 1.35 for the factorization and 156/113 = 1.38 for the forward and back substitution.

4. Assembly-language programming

The speed of an algorithm is related to the language in which it is programmed. Thus, it is important to document the extent to which an algorithm is influenced by programming at least the inner loop(s) in a high level language such as Fortran.

An assembly language version of the expanded-column factorization algorithm of Figure 4a was written, with the inner two loops (k=1,2,...r-1) of Eq. (12) coded in assembly language. Table A3 gives timing comparisons vis-a-vis a Fortran H implementation. For a 961 x 961 finite element matrix, the ratio

$$\frac{\text{Fortran H timing}}{\text{Assembly timing}} = \frac{1309}{922} = 1.41$$

shows a significant but not atypical advantage in machine-dependent coding. This depencence on the quality of Fortran code must be

minimum will in general be quite broad. The sharpest min-

borne in mind when other timing comparisons are made in this report.

F. Partitioning

1. Introduction

Partitioning the solution of simultaneous linear equations refers to the division by the user of matrix and/or LU storage between a local store (either real or virtual) and a slower backing store accessible to the ALU only through the local store. The local store contains that part of the matrix on which computation is being performed, so that read/write (I/O) operations are necessary between local and backing store to carry out the complete matrix factorization. The purpose of this partitioning is either (1) to meet the storage limitations imposed by a real memory too small to contain the complete factored matrix, or (2) to minimize the cost of computing in a real or virtual system, where costs depend on both storage used and user-prescribed I/O operations. The sparse matrix software to be described in the next chapter has considerable flexibility in performing this partitioning even before the matrix is numerically solved. Wise use of this flexibility requires modeling both the computing system and the matrix structure. The purpose of the following discussion is to provide insight into the issues involved and, by example, specific guidelines for the user of the software.

Before proceeding, it is worth noting that many of the concepts and even some of the detailed analysis to be presented applies to other memory heirarchies such as register-cache, register-main, and cache-main, where a small fast memory communicates with a large slow memory.

2. Fixed local store

a. Introduction

The simplest model for analysis is that of a processor with fixed local store. In this environment, common to large scientific processors, the I/O operations are managed by either (a) the central processor, or (b) a peripheral processor sharing the local store with the central processor. In the first case, the total CPU time would be the dominant issue; in the latter, the ratio δ = (I/O time)/(CPU time) would yield the fraction of time that the I/O processor is busy, so that when $\delta > 1$ the CPU must wait on the I/O processor for its operand supply. This section will concentrate on analysis of the δ ratio, since the former will be a special case of variable storage time minimization (b=0) considered in the next section.

Clearly, both the CPU time and storage depend on matrix structure so that an analysis must be aimed either at obtaining a precise evaluation of a given matrix structure or at establishing general relationships for a class of matrix structures. In preparation for the latter, the specific case of a full matrix will be studied to gain insight into the analysis procedure. The reader may refer to [17] for alternate partitioning schemes that may result in less I/O traffic for full matrices than the one to be studied here.

b. Full matrices

In Figure 5, a full matrix of size n (=6) is divided into k partitions of size S_k, each having p = n/k columns. The number of writes to backing store is n^2, assuming that the entire factored matrix must reside on backing store. The number of reads is

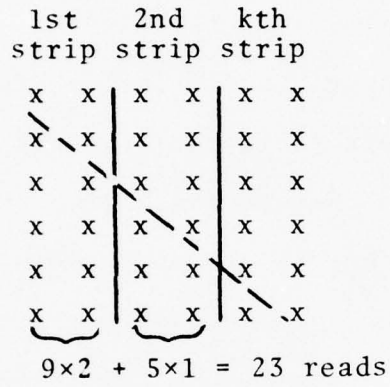


Figure 5. Counting reads in a factorization

$$N_r = \sum_{r=1}^{k-1} \frac{p(p-1)}{2} r + p^2(k-r)^2$$

$$= \frac{p(p-1)(k-1)(k)}{4} + \frac{p^2(k-1)(k)(k-\frac{1}{2})}{3} \qquad (20)$$

With total storage S and strip size S_k, then $k = S/S_k$ and $p = S_k/\sqrt{S}$ The second term of (20) easily dominates the expression, and the dominant term can be written

$$N_r \cong \frac{p^2k^3}{3} = \frac{S^2}{3S_k} \qquad (21)$$

If the I/0 transfer time is c seconds/byte and the floating point operation time is d seconds/(multiply-subtract), the ratio

δ is

$$\delta = cN_r/dN_{op} \quad . \tag{22}$$

With typical values* $c = .23\text{x}10^{-6}$ and $d = 2.24\text{x}10^{-6}$ and assuming 64-bit words, for large n

$$\delta = [(8n^2)^2(.23\text{x}10^{-6})/(24\ S_k)]/[2.24\text{x}10^{-6}n^3/3]$$

$$= .824n/S_k$$

Thus, when $n/S_k = 1$ (only one column is in local store), a nearly equal time is devoted to arithmetic computation and to I/O. For S_k larger than this absurdly small value, arithmetic time predominates.

For a vector processor such as the Cray 1, where $d = 12.5\text{x}10^{-9}$ and $c = 1.56\text{x}10^{-8}$, the ratio becomes

$$\delta = 10.n/S_k$$

and a local store of 10 columns would be adequate to keep the processor supplied with operands.

c. Sparse matrices

For families of sparse matrices with a relatively constant column density, the storage of $\underline{L}$ and $\underline{U}$ is typically $S = 0(n\log n)$ or even $S = 0(n)$ and the computation time is typically $0(n^{1.5})$, in contrast to $0(n^2)$ and $0(n^3)$ respectively for a full matrix. Thus, the asymptotic ratio of computation to storage is lower for a sparse matrix, hinting that the I/O transfer may become a more significant cost factor. It will be shown that, asymptotically in n, I/O can be a <u>less</u> significant factor for sparse matrices; further, some guidelines will be established for the estimating the growth with n.

The full matrix model of Figure 5 may be generalized by considering a matrix of k partitions, each with $S_k = S/k$ storage. This does not require that each partition have the same number of

*For the Amdahl 470V/6 system at the University of Michigan.

columns as above, allowing more columns in the usually sparse initial column strips. As shown in Figure 6 the $\underline{L}$ storage is assumed distributed throughout each strip so that

a) $S_{k,L} = S_{L_o} + (k-r)S_{L_1}$; $S_{k,U} = S_{U_o} + (r-1)S_{U_1}$ $r = 1,2,\ldots k$

b) the non-zero structure of each column strip is distributed so that each strip must be recalled* for the factorization of all succeeding strips.

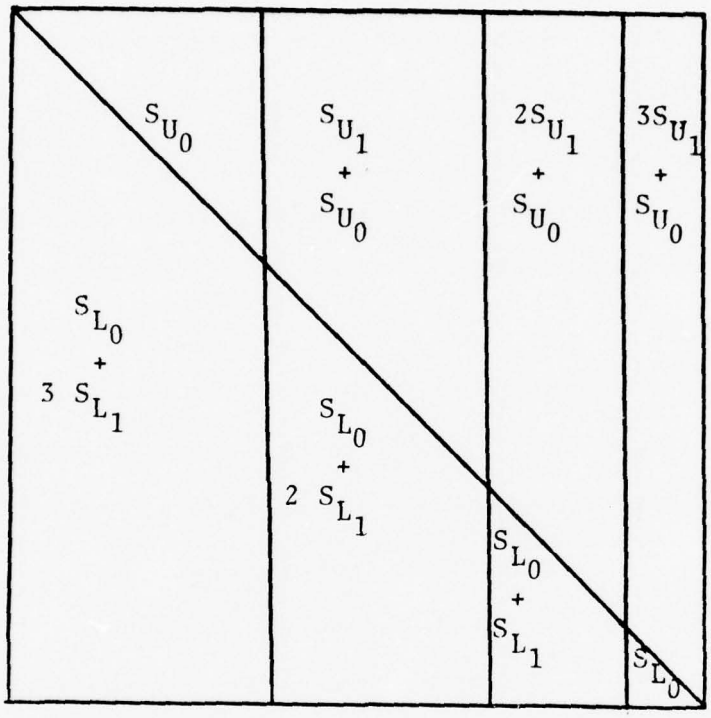


Figure 6. Sparse matrix partition

The total storage S is then given by

$$S = \sum_{r=1}^{k} (S_{L_o} + S_{U_0}) + (k-r)S_{L_1} + (r-k)S_{U_1}$$

$$= k^2 S_1 + 0(k) \qquad (23a)$$

where $S_1 = S_{L_1} = S_{U_1}$. The recalled storage is then

*It will be assumed throughout this report that only complete $\underline{L}$ strips can be recalled.

$$N_r = \sum_{r=1}^{k-1} (S_{L_o} + rS_{L_1})r$$

$$= \frac{k^3}{3} S_1 + 0(k^2) \qquad (23b)$$

Eliminating S_1 from (23a) and (23b) gives

$$N_r = \frac{kS}{3} = \frac{S^2}{3S_k} \qquad (24)$$

the same as (21) for a full matrix.

To experimentally corraborate this simplified expression, Table 6 shows the total number of reads of symbolic and numeric information during the factorization of the finite element matrices of appendix Table A4, together with the value calculated from (24). The error is within 10 percent for large matrices, the discrepency probably due to assumption that all strips must be recalled in the factorization of the current strip.

S_k	$S^2/3S_k$	Experimental
.232	.13	.15
.101	.31	.22
.0355	.88	.83
.006500	4.8	4.25

Table 6. Comparison of calculated and experimental recall reads (in megabytes). S = 306,000 bytes.

As a result of (24), for a constant S_k, for S = 0(n log n), 0(n)

$$N_r/N_{op} = 0(n^{.5}(\log n)^2),\ 0(n^{.5}) \qquad (25)$$

indicating that relatively fewer reads are required for a sparse matrix than for a full matrix, for sufficiently large k and n. A more qualitative analysis will shortly show an excessive number of reads <u>can</u> occur for typical values of n.

Alternatively, Equation (25) can be used to estimate the growth of local storage necessary to maintain a prescribed N_r/N_{op} in the event I/0 does become significant. For the model, with S = 0(n),

$$N_r/N_{op} = 0(n^{.5})/S_k \qquad (26)$$

so that a slow growth in local storage will maintain a constant ratio.

A key assumption in the above model was that all preceding strips had to be recalled for current strip factorization. At the other extreme, each strip could require the recall of only one preceding strip, as in an appropriately-partitioned band matrix (Figure 7). It is easy to show for such a case that $N_r/N_{op} = 0(1)$. Tridiagonal matrices - forms of band matrices - clearly have a similar dependence.

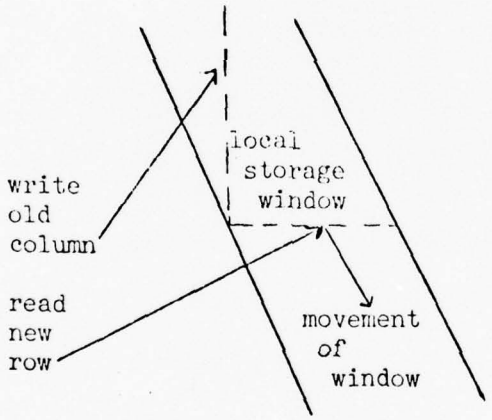


Figure 7. Banded solution

In general, sparse matrices have a structure that varies between local and complete strip connectivity. Thus, with a fixed local store, the growth N_r/N_{op} would be expected to be between 0(1) and $0(n^{.5})$.

One caveat must be raised. Sparse matrices are inevitably ordered so as to reduce fills occuring during solution. Ordering algorithms can not be expected to detect strip connectivity, and it is not uncommon for strips to have widely-distributed coupling to other strips. Thus, a strip storage equalization as illustrated in Figure 8a may not yield a minimum number of I/0 operations, due to scattered non-zero positions. A slight adjustment yields the improved situation of Figure 8b.

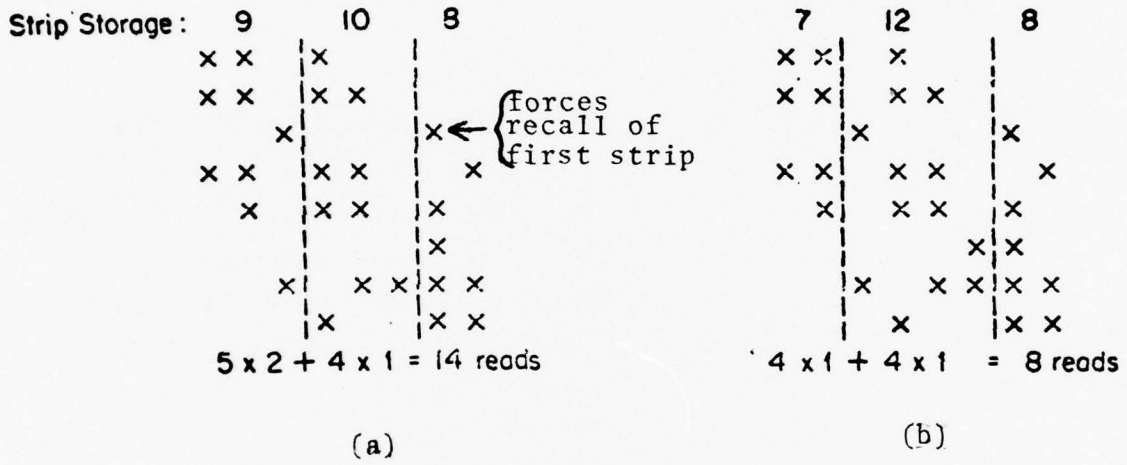


Figure 8. Reduction in reads by unequal strip storage.

d. The I/0 problem for finite element solutions

Although asymptotically the I/0 problem may be less serious for a sparse matrix, the more important issue is whether I/0 will dominate for a particular sparse matrix. An illustrative analysis for the finite element family of Appendix A will show that a serious problem can indeed exist even for the computationally-intensive factorization process.

From the asymptotic formula for storage count in Table A1, the read time can be counted as

$$T_r = T_{tr}(S^2/3S_k)$$

$$= T_{tr}(124n-344)^2 2^{4n}/24S_k \qquad (27)$$

where T_{tr} is the transfer time of a 64-bit word. The arithmetic operation time is

$$T_{op} = T_{ar}(39.5(2^{3n}))$$

where T_{ar} is the time of an arithmatic operation. Now define the critical dimension as that value of $n(=n_c)$ for which $T_{rw} = T_{op}$; for $n > n_c$, T_r will exceed T_{op}.

	I/O Transfer Time (sec./word)	Local Store (megawords)	MFLOPS	Grid size	Storage (S) (megawords)	Time (sec.)
ASC (1-pipe)	2×10^{-7} - 10^{-6}	1 - 8	25.	132 - 1606	1.1 - 315.	7.3 - 13100.
ASC (4-pipe)	2×10^{-7} - 10^{-6}	1 - 8	100.	60 - 588	.172*- 34.	.16 - 160.
Cray 1	10^{-7}	1 - 4	160.	175 - 430	2.2 - 17.1	2.6 - 39.4

*Backing store not required

Table 7. Grid size for equal I/O and operation times on current vector processors

Table 7 shows that the critical dimension can be surprisingly small for current vector processors. Ranges for the critical grid size ($\cong 2^{n_c}$) are given for normal ranges of T_{tr} - corresponding to commonly used disc capacity - and of S_k. An interesting case is the Cray-1, where because of extraordinary processor speed, the I/O time will exceed the arithmetic operation time after 39.4 seconds, even using full I/O channel capacity and addressable local store.

e. The I/O problem in the substitution process

The I/O problem is proportionally more severe for the forward and back substitution steps, where only a few numeric computations are performed with each element of the recalled factored matrix.

Consider an arithmetic computation sequence where K arithmetic computations are performed on the average on every L words in memory, by a processor with an operation rate of M floating point operations/second (FLOPS). Then the local store must be supplied from the backing store at the rate of

$$ML/K \text{ words/sec.} \tag{28}$$

In the forward and back substitution stages, the inner loop instruction will be of the general form

$$X(I) = X(I) - LU(J)*YD \tag{29}$$

where LU(J) contains the elemtns of L and U. Each such element is used a single time in the two substitutions, so that (ignoring array X and scalar YD) L = 1 and K = 2 in (28). If the LU array is on a backing store, then this store is required to supply operands for (29) and M/2 words/sec.

This is a prohibitive rate, as evidenced in Table 8. Here, the processor utilization ratio given by

$$PUR = \frac{\text{arithmetic operation time}}{\text{I/O + arithmetic operation times}}$$

are given for several of the current vector processors executing the substitution formula of (29). The PUR is susally below .15 and becomes as low as .02.

It is important to note that this poor processor utilization is dependent only on the machine system parameters, and not on the matrix size, density, or equation ordering. Thus, the concern

of Knight et al [19] for banded matrices is shown to be generalized to the substitution process itself.

Processor	I/O Transfer Time (sec./word)	Mult.-Sub. Time (sec./op.)	PUR
ASC (1-pipe)	2×10^{-7} - 10^{-6}	80×10^{-9}	.28 - .074
ASC (4-pipe)	2×10^{-7} - 10^{-6}	20×10^{-9}	.091 - .019
Cray 1	10^{-7}	12.5×10^{-9}	.11

Table 8. Projected processor utilization for current vector processors.

To support this claim of independence of matrix properties, the experiments of Table 3 for the Amdahl 470V/6 scalar processor can be used. The operation time of the inner loop has been determined in Table 4 ($=2.24\times10^{-6}$ sec/operation), and the I/O transfer time given in Table A4 as 1.83 sec./word. The PUR is then calculated as for all matrices as

$$PUR = \frac{2.24}{2.24 + 1.83} = .55$$

Experimentally, the same ratio can be determined from the components of T_{FBS} of Table A4 for the finite element family of matrices.

Matrix size	PUR
49	.61
225	.58
961	.56

Table 9. Illustration of constant processor utilization ratio.

The results are given in Table 9; these show a PUR close to .55 for all matrices, and an asymptotic approach to this value for the larger matrices. The small descrepency is due to the loop startup and pivoting times not included in the above model.

At present, there is no known solution to this problem in the substitution process. Either one must hope that the processor can be utilized for other tasks while the I/O is busy, or, in some (few) applications, multiple substitutions may be carried on simultaneously with the same factorized matrix.

3. Variable local store

Local store is commonly variable as a result of a multi-user environment, i.e., a large store is divided between two or more users. If this large local store is virtual, then it will be assumed that the user is not assessed the costs of paging or swapping from the system's backing store.

When I/O is handled by the central processor, the cost of variable local store is commonly charged according to the length of time and the amount of store used. Specifically, the cost of a program execution can be written

$$\begin{aligned} \text{cost} &= a\ (\text{CPU time}) + a\ b\ (\text{local storage})(\text{CPU time}) \\ &= a\ (1 + b\ (\text{local storage}))(\text{CPU time}) \end{aligned} \tag{30}$$

where a is a charging coefficient (dollars/CPU-second)

b is a coefficient (pages^{-1}) that converts the storage costs to unit CPU costs.

The total CPU time is clearly $c\ N_r + d\ N_{op}$. The local storage must have a value $2\ S_k$ to accomodate both the current strip and the recalled strip. The cost can then be written as a function of S_k for a full matrix solution with large n as

$$\begin{aligned} \text{cost} &= a(1 + b(S_p+2S_k))(\frac{cS^2}{3S_k} + dN_{op} + T_{ov}) \\ &= \frac{k}{S_k}(\frac{1+bS_p}{2b} + S_k)(\frac{cS^2}{3(dN_{op}+T_{ov})} + S_k) \\ &\simeq \frac{k}{S_k}(\alpha_1 + S_k)(\alpha_2 + S_k) \end{aligned} \tag{31}$$

where S_p is the fixed program storage and T_{ov} is overhead time not attributable to either numeric or I/O computation. Although this cost function has a minimum at $S_k = \sqrt{\alpha_1\alpha_2}$, the

minimum will in general be quite broad. The sharpest minimum occurs when $\alpha_1 = \alpha_2$ or n = 41000 and S_k = 50 pages, for b = .01. Even this minimum is broad; the cost does not exceed double the minimum value for $\alpha_1/7 < S_k < 6\alpha_1$. For more practical values of n, the local storage costs considerably exceed the I/0 costs, and the former can easily be evaluated from the first factor in (31).

To apply this analysis to a sparse equation system for which (31) applies, consider again the finite element family of Appendix A, solved on the University's system. A summary of parameters and formulae pertinent to this calculation are shown in Table 10.

Program storage S_p (pages)		
	Driver with arrays	7.7
	Solver with I/O	7.0
	System routines	7.8
	I/0 buffers	43.0
	Total	65.5
LU storage S (Table A1)		$(124n-344)2^{2n}$
T_{ov}		0.0 sec.
Arithmetic operations (N_{op})		$39.5(2^{3n})$
I/0 transfer rate (1/c)		1000 pages/sec.
Arithmetic operation time in inner loop (d)		1.1×10^{-6} sec/operation

Table 10. Parameter summary

For n=5, a 961 equation system, it may be determined that α_1 = 82.5, α_2 = 1.09, yielding a cost minimum at S_k = 9.5 pages. The estimated cost is plotted versus S_k in Figure 9; the minimum is shown to be quite broad due to the relatively

small cost of local store on this virtual memory system (e.g., b=.01 implies that when $S_p + 2S_k = 100$ pages, storage costs become equal to CPU numeric computation costs). Several correlations of estimated and actual solution costs has been made. Figure 9 shows a rather large discrepancy due principally to the use of inner loop timing estimates only, the failure to include substitution steps in the estimate, and the assumption that $T_{ov} = 0$. Note that Table A4 shows that, for $2S_k$ = 13k bytes, approximately .57/3.7 = .15 of the factorization time is unaccounted for. It may be that this overhead is compensated by the use of asymptotic values for S in Table 10.

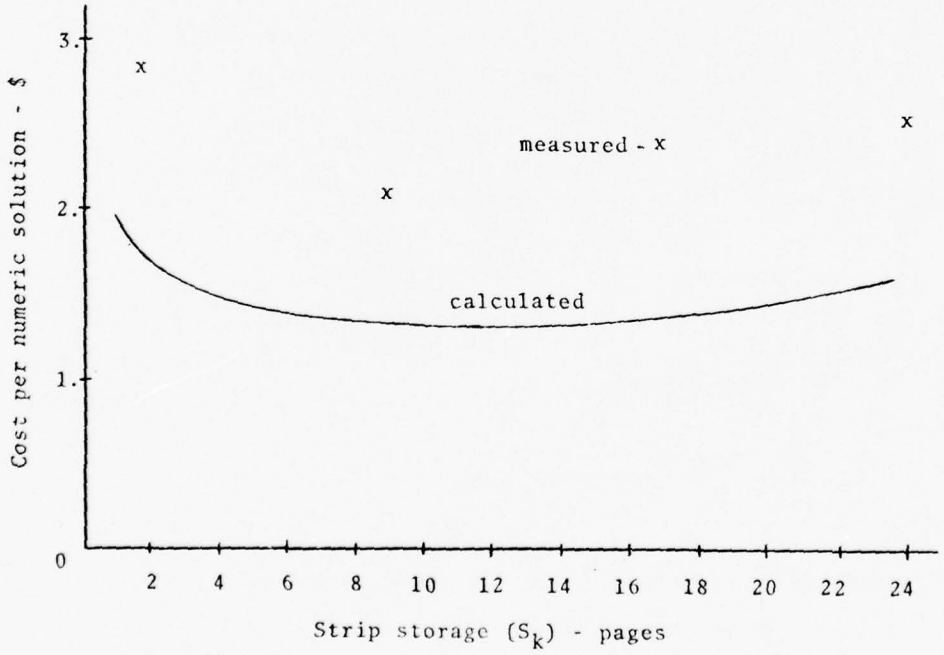


Figure 9. Comparison of measured, calculated cost of virtual memory numerical solution.

G. Evaluation

This chapter has been primarily concerned with modeling and analysis of components of the equation solver. The detailed timing analysis performed on critical parts such as arithmetic kernels and I/O routines cannot reasonably be performed on every code sequence. Therefore, although the asymptotic performance for large problems may be reliably predicted from these key components, the small problem performance cannot; indeed, one does not know for certain what qualifies as a small problem without analysis of all the code segments. Fortunately, since the complete equation solvers have been implemented (Chapter 3), it is possible to make evaluations by comparing run time performance.

One high-level performance measure frequently used for evaluation of processors with complicated parallel/pipeline architectures executing simple software kernels is the millions of floating point operations per second (MFLOPS). Plotted versus problem size, the rate at which the MFLOPS approach asymptotic values imposed by the speed of processor arithmetic units gives a succinct display of overall small problem performance meaningful to user and algorithm developer alike. We propose to use the same MFLOPS performance as a comparative measure of complicated scientific packages executing on a relatively simple (scalar) architecture.

Figure 10 shows the MFLOPS dependence of four sparse equation solvers applied to the family of finite element problems of Appendix A. The asymptotic values shown are

obtained from the execution mode timings of the innerloop obtained in Table 4. The MFLOPS are obtained from the timings of Table A3 and a knowledge of computational complexity. Several comparisons of the plots seem meaningful.

1. Over the range of n shown, the scalar version shows a relatively superior performance over the vector version for intermediate values (n=3,4) where the overhead of the inner loop is significant. For n=2, other program overhead tends to produce equally poor performance for both versions; for n=5, the larger inner loop startup time of the vector version becomes less significant and the gap between the two decreases.

2. The large program overhead of the partitioned version makes this very unattractive for n=2,3, but when n=5 the rate of increase in MFLOPS is large and suggests that this version should be operating at 50% or more of the asymptotic MFLOPS value for problems beyond the range of a 1-2 megabyte local store.

3. The code generation version shows a remarkably fast climb to the asymptotic value (the vector inner product execution timing of Table 4 is used). Unfortunately, the code length did not permit larger values of n to be investigated.

It is planned to extend these revealing characteristics to larger values of n for the partitioned version, and to test this version on commercial vector processors. Perhaps more important, this appears to be a useful measure of the efficiency of general sparsity algorithms versus more specialized full-, band-, and block-solvers operating from a backing store. If the general algorithm can be shown to be as easy to use (see formulational aids, Chapter 3) and near the MFLOPS rate of special algorithms, the usefulness of the partitioned general solver will be well established.

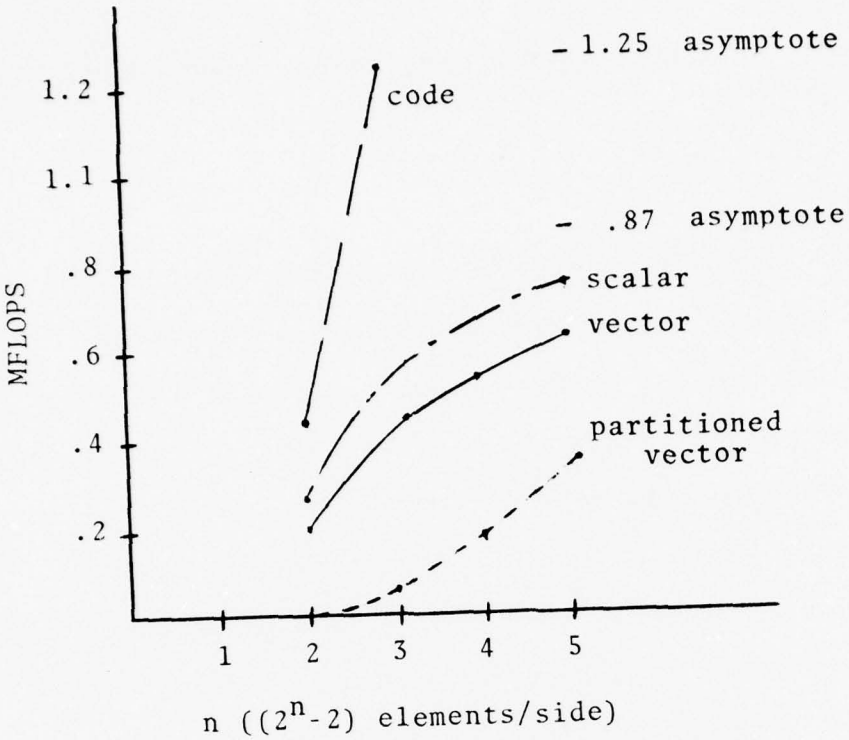


Figure 10. Floating point operation rate

Chapter 3. DESCRIPTION OF A DUAL VECTORIZED GENERAL SPARSE EQUATION SOLVING PACKAGE

A. Introduction

The results of the appendix referenced throughout Chapter 2 were produced by two general sparse equation solvers. Together with the (processor-dependent) code generation program of [6] [8], these Fortran language solvers allow the user to solve sparse sytems of equations ranging from tens to tens of thousands of eqations. For small systems, Table A3 shows that code generation is several times faster than other algorithms; when memory size to contain the code becomes a limitation - perhaps for several hundred equations - the matrix often has sufficient local density to warrant a vectorized solution. As the system size increases to several thousand equations the LU storage becomes excessive and a program with backing store becomes necessary. Since the majority of sparse matrices solved on current fast scalar and vector processors are beyond the size manageable with code, the latter two approaches appear the most useful.

The dual software package described in this chapter allows the user to begin with an equation solver contained in local store and then, as problem size grows, to include a backing store with little change to the application program. There is, of course, some reprogramming of the I/O section of the second solver necessary for different file systems.

In developing the dual sparse solver package, several general guidelines were used.

1) The package should be able to solve efficiently sparse equations ranging in density from several elements per column

(electrical power systems) to several hundred elements per column (finite element problems). Since the structures of such matrices are described by quite different data structures, this assumption resulted in a distinction being made between the user-supplied matrix description, and the data structure utilized by the symbolic and numeric solvers. Thus, any of a variety of structural descriptions can be entered by the user (see flow chart of Figure 15); these are converted to a common vectorized data structure prior to symbolic and numeric processing. It is recognized that an additional storage will be required to describe the structure of block matrices, vis-a-vis a "hypermatrix" [14][15][16] representation. However, as shown in Table A1, even for relatively small sparse matrices, the symbolic storage is far less than the numeric storage - especially for machines with 16-bit integer format - so that the additional I/O incurred in recalling previous column strips containing both numeric and symbolic imformation is not significant.

2) The processor should have at least 1-2 megabytes of local (real or virtual) store. Thus, it is assumed that the symbolic matrix structure fits in local store and that a full column (row) of the numeric storage occupies a small fraction of local store. This assumption rules out a typical minicomputer system.

3) The package should run efficiently on both scalar and current vector processors, although it is recognized that performance could be improved in some combinations of matrix structures and vector architectures by major revision of the symbolic and numeric algorithms (e.g., blocked matrices being solved on a processor with a high-level vector capability, or very sparse matrices

being solved by multi-row factorization [13]). Certain common vector instructions are accomodated (e.g., the inner-product instruction) by supplying alternative forward and back substitution algorithms.

In this chapter, issues related to the use of the package are examined, including

(1) a simple example of its use, without backing store and with user vectorization of the structural input data;

(2) discussion and examples of aids to simplify the equation formulation and avoid the need for vectorization noted in (1);

(3) general flow chart of the program;

(4) representative example involving finite elements and using backing store and formulation aids;

(5) detailed discussion of algorithms and formats used in the sparse solvers.

B. General Program Description and Use

1. Program description

The software package is divided into three operationally distinct parts.

(1) VEGES (VEctorized General Equation Solver), operating without a backing store;

(2) VEGES/P, which partitions the matrix solution to utilize backing store, and

(3) UNBLOK, a symbolic preprocessor that accepts a variety of user descriptions of the matrix structure and produces vectorized arrays to VEGES or VEGES/P to aid the numerical equation formulation procedure.

The specifications for VEGES and VEGES/P are given in Table 11. It is worth noting VEGES/P requires considerably more program storage than VEGES, so that the latter should be used when partitioning is not required.

2. Example use of VEGES

For the reader unfamiliar with the use of sparse equation solvers, an elementary example will depict a minimal programming effort necessary to utilize this software. To distinguish the symbolic and numeric solution phases, separate main programs are used for each phase; communication between phases is provided through a backing store. Since the symbolic phase need be executed only once for a given matrix structure, this separation insures that program and array storage associated with only the symbolic phase will not burden the time-consuming numeric phase.

1. Name: VEctorized General Equation Solver (VEGES without partitioning, and VEGES/P with partitioning).

2. Purpose: To perform direct solution of arbitrarily-structured sparse simultaneous linear equations, either with or without a backing store (partitioning).

3. Language: IBM extended Fortran (see IBM document GC28-6515-10); principal extensions from ANS Fortran are IMPLICIT, REAL*8, and INTEGER*2 declarations.

4. Operating system: Development and testing performed on Michigan Terminal System.

5. Availability: Source language programs available from Professor Calahan on 9-track, 800/1600/6250 BPI (1600 default), IBM standard labeled (default) or unlabeled magnetic tape.

6. Program limitations: up to 32768 equations; this limit can be changed by altering INTEGER*2 array types to INTEGER*4.

7. Program storage (bytes) VEGES: Symbolic - 5718
Numeric - 3976
VEGES/P: Symbolic - 18250
Numeric - 15300
I/O management routines - 12456

Table 11. Program specifications.

The example solved by the program of Table 12 is

$$\begin{bmatrix} 1. & 2. & 1. \\ 0. & 5. & 2. \\ 2. & 0 & 5. \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8. \\ 16. \\ 17. \end{bmatrix} \qquad (32)$$

where $[x_1\ x_2\ x_3] = [1\ 2\ 3]$ is to be found using (1,3),(2,2), (3,1) as pivot positions. Beginning with the column-ordered vectorized matrix structure (recall Table 2)

IA: -1,-3,1,2,1,3

JA: 1,3,5,7

and the pivot positions

IPC: 3,2,1 IPR: 1,2,3.

first the order of IPR is inverted (IPRI(IPR(J))= J and the symbolic processor VMSP produces a set of seven arrays passed to the numeric phase. In the numeric solution, these are combined with the column-ordered matrix and right-hand side

A: 1.,2.,2.,5.,1.,2.,5.

B: 8.,16.,17.

in VMNP and VMBPC to produce the solution. The flow chart is shown in Figure 11.

It should be noted that, although the example depicts column-ordered structure, row ordering can be accomodated by using subroutine VMBPR for the substitution steps. The symbolic and numeric data would have the row-ordered form

IA: 1,3,2,3,-1,-3,

JA: 1,3,5,7,

IPR: 3,2,1, IPC: 1,2,3,

A: 1.,2.,1.,5.,2.,2.,5.

```
C**** SYMBOLIC PHASE
C**** DIMENSIONS: IVA(NP1),JU(NP1),JL(NP1),IVU(NP1),IVL(NP1),IP(N)
C      IXT(N),IXB(N),IPR(N),IPRI(N),IPC(N),ICNT(N,7),JA(NP1)
C      DIMENSION OF IA = NO. OF POSITIONS IN MATRIX
C      DIMENSION OF JU,JL = NO. OF POSITIONS IN U,L
       INTEGER*4 JA(4)/1,3,5,7/,IVA(4),JU(4),JL(4),IVU(4),IVL(4)
       INTEGER*2 IA(6)/-1,-3,1,2,1,3/,IU(9),IL(9),ICNT(3,7),
      &IP(4),IXT(4),IXB(4),IPC(3)/3,2,1/,IPR(3)/1,2,3/,IPRI(3)
       NDSRN=1
       N=3
       MAXS=9
       MAXN=9
C**** INVERT ROW PIVOT VECTOR
       CALL PIVI2(IPR,IPRI,N)
C**** CALL SYMBOLIC PROGRAM TO GENERATE MAP
       CALL VMSP(N,JA,IA,IVA,JU,JL,IU,IL,IVU,IVL,IXT,IXB,IP,
      &ICNT,MAXS,MAXS,MAXN,MAXN,N,IPC,IPRI)
       NP1=N+1
C****  DIVISION BY 2 FOR *2 ARRAYS
       I3=(JU(NP1)+1)/2
       I4=(JL(NP1)+1)/2
       I5=(JA(NP1)+1)/2
C**** WRITE ARRAY DIMENSIONS FOR NUMERICAL SOLUTION
       WRITE(6,2)IVU(NP1),IVL(NP1),JU(NP1),JL(NP1),JA(NP1)
 2     FORMAT(' U-',I5/' L-',I5/' IU-',I5/' IL-',I5/' MATRIX-',I5)
C**** SAVE SYMBOLIC RESULTS ON DISC (FOR ILLUSTRATION PURPOSES)
C**** WRITE ARRAY LENGTHS
       WRITE(NDSRN,3)I3,I4,I5
 3     FORMAT(3I5)
       CALL WRT(JA,NP1,NDSRN)
       CALL WRT(IA,I5,NDSRN)
       CALL WRT(IVA,NP1,NDSRN)
       CALL WRT(JU,NP1,NDSRN)
       CALL WRT(JL,NP1,NDSRN)
       CALL WRT(IU,I3,NDSRN)
       CALL WRT(IL,I4,NDSRN)
       CALL WRT(IVU,N,NDSRN)
       CALL WRT(IVL,N,NDSRN)
       CALL WRT(IPC,N,NDSRN)
       CALL WRT(IPRI,N,NDSRN)
       STOP
       END
       SUBROUTINE WRT(I,NTOT,NDSRN)
C**** ILEN IS LENGTH OF WRITE IN 4-BYTE WORDS
       INTEGER*2 ILEN
       LOGICAL*1 I(1)
       NTOT1=NTOT*4
       NBASE=1
     1 NTOT2=MIN0(NTOT1,32000)
       ILEN=NTOT2
C**** SEQUENTIAL WRITE...32000 BYTES PER RECORD
       CALL WRITE(I(NBASE),ILEN,0,IDUMM,NDSRN)
       NTOT1=NTOT1 -32000
       NBASE=NBASE +32000
       IF(NTOT2.EQ.32000) GO TO 1
       RETURN
       END
C**** NUMERIC PHASE
C**** DIMENSION U,L,IU,IL,IA PER PREPROCESSOR PRINTOUT
       REAL*8 A(7)/1.D0,2.D0,2.D0,5.D0,1.D0,2.D0,5.D0/,U(4),L(4),
      &DI(3),B(3)/3.D0,16.D0,17.D0/,X(3)
       INTEGER*4 JA(4),IVA(4),JU(4),JL(4),IVU(4),IVL(4)
       INTEGER*2 IA(7),IU(4),IL(4),IPC(3),IPRI(3)
       NDSRN=1
       N=3
       NP1=N+1
C**** READ DATA FROM PREPROCESSOR
       READ(NDSRN,1)I3,I4,I5
 1     FORMAT(3I5)
       CALL RDE(JA,NP1,NDSRN)
       CALL RDE(IA,I5,NDSRN)
       CALL RDE(IVA,NP1,NDSRN)
       CALL RDE(JU,NP1,NDSRN)
       CALL RDE(JL,NP1,NDSRN)
       CALL RDE(IU,I3,NDSRN)
       CALL RDE(IL,I4,NDSRN)
       CALL RDE(IVU,N,NDSRN)
       CALL RDE(IVL,N,NDSRN)
       CALL RDE(IPC,N,NDSRN)
       CALL RDE(IPRI,N,NDSRN)
       CALL VMNP(N,JA,IA,IVA,A,JU,JL,IU,IL,DI,U,L,X,IVU,IVL,
      &IPC,IPRI)
       CALL VMBPC(N,JU,JL,IU,IL,IVU,IVL,DI,U,L,B,X,IPC,IPRI)
       WRITE(6,2)(B(J),J=1,N)
 2     FORMAT(3E15.7)
       STOP
       END
       SUBROUTINE RDE(I,ILEN,NDSRN)
C**** ILEN IS LENGTH OF READ IN 4-BYTE WORDS
       INTEGER*2 IXEN
       LOGICAL*1 I(1)
       NB1=ILEN*4
       NBASE=1
     1 NB2=MIN0(NB1,32000)
       IXEN=NB2
C**** SEQUENTIAL READ
       CALL READ(I(NBASE),IXEN,0,IDUMM,NDSRN)
       NB1=NB1 -32000
       NBASE=NBASE + 32000
       IF(NB2.EQ.32000) GO TO 1
       RETURN
```

Table 12. Fortran program to solve Equation 32 using VEGES.

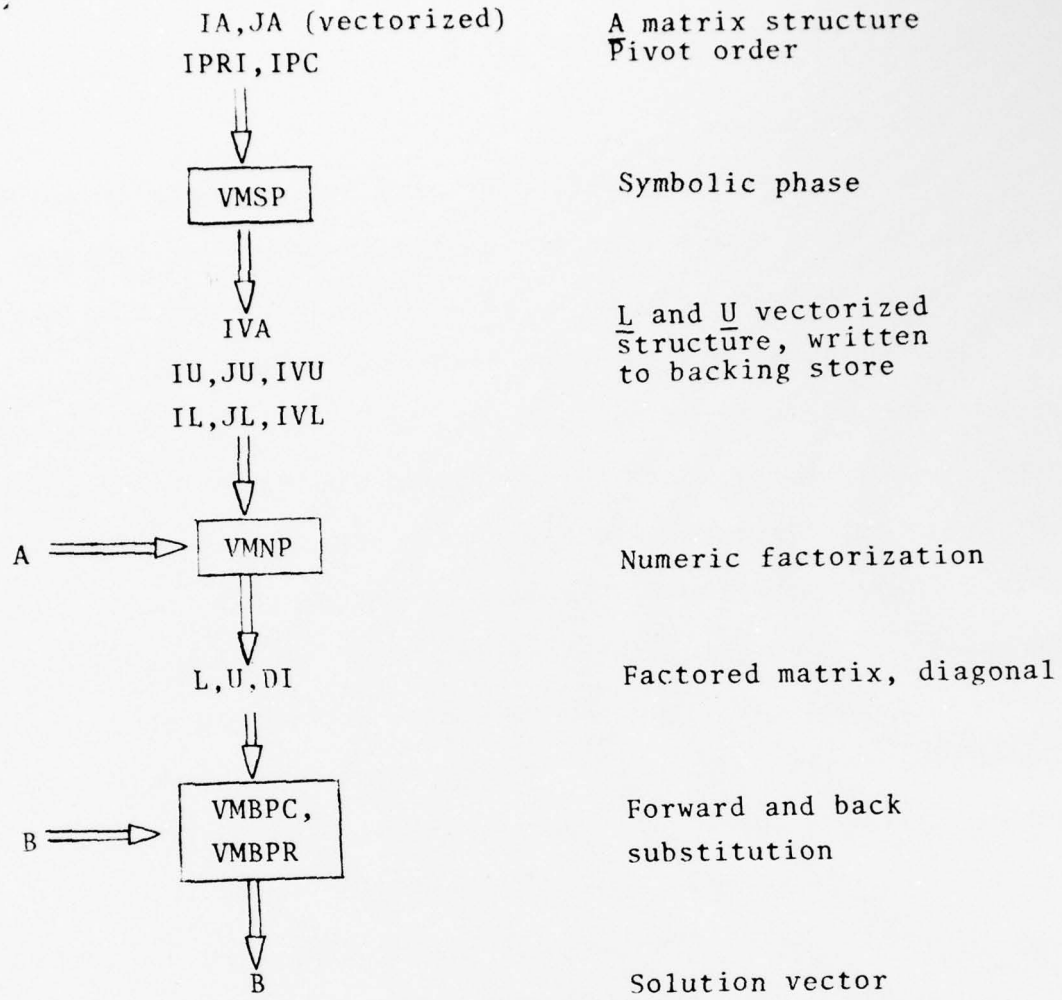


Figure 11. Flow chart of non partitioned vectorized sparse matrix solver.

C. Simplified Equation Formulation

1. Introduction

From a formal viewpoint, a general sparse equation solver can reasonably require that the matrix be stored in standard order (e.g., row or column) in local or backing store before the solution is initiated. The burden of formulating the equations in this order is left with the user, so this rationale goes, because the variety of equation-formulation procedures is simply too large to accomodate in a general way. Unfortunately. this standard-ordering requirement, together with the unavoidable necessity of describing the sparsity structure, has made the conversion from full- and band-oriented methods to general-sparsity methods worthwhile only for large problem-oriented analysis/design packages where the user is isolated from the demands of the sparse equation solver.

Several aids to the equation formulation have been produced to avoid this column-ordering requirement on the user. Their use is illustrated in the following examples. In this development, a distinction will be made between a scalar structure, where individual elements of the matrix are described by their (row, column) position, and a vector structure, where submatrices (blocks) are indicated by several descriptors.

2. Scalar case: example

To exemplify a typical conversion from a full- (or band-) matrix solution to a general sparsity solution using the formulation aids, consider the elementary example of Figure 12 (the reader is assumed to be familiar with the example of Table 2). The key steps in this process are the following.

numeric factorization portion of VEGES.

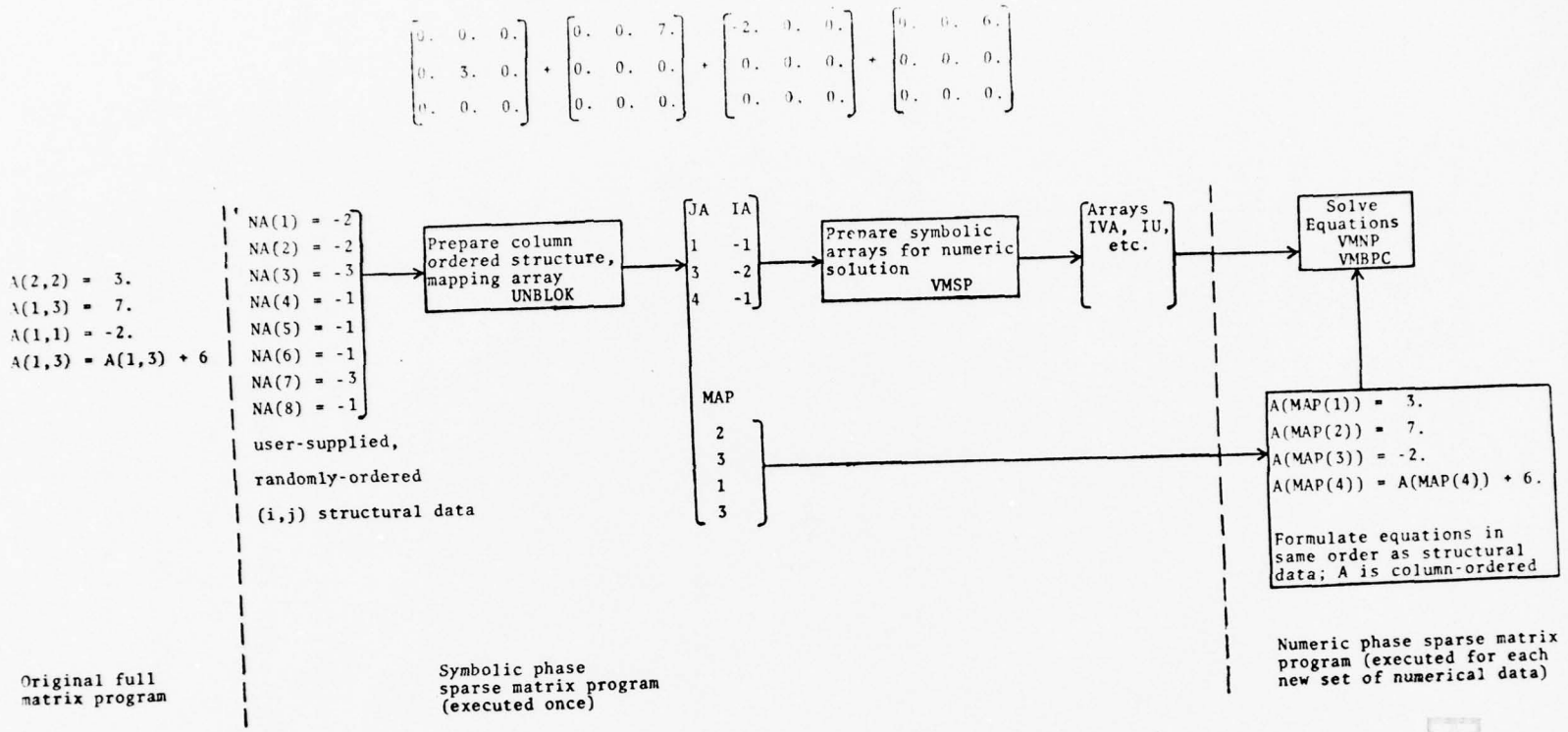


Figure 12. Comparison of full matrix, sparse matrix formulation and symbolic processing.

at least an order of magnitude less storage than numeric arrays

1) The user must supply explicit structural data, but the structure may be in any order and in a variety of forms; the scalar (i,j) form is illustrated in Figure 12. This structure is most easily produced by creating a new version of the full matrix formulation, but replacing each numerical formulation step with a symbolic formulation step (a four-step formulation is shown); this symbolic data (array NA) is preprocessed in the symbolic phase of Figure 12.

2) Because the sparse solver requires column (row) ordering of numerical data, but the equations in general may be formulated in any order, a mapping array MAP is generated by subroutine UNBLOK, with an element for each numeric formulation step, so that at the ith formulation step, A(MAP(I)) is calculated by the user. Alternatively, if the numeric values are stored in an array B (viz, B(1) = 3., B(2) = 7., etc), the A matrix array is formed by

```
      DO 1 J=1,4
    1 A(MAP(J))=A(MAP(J))+B(J)                    (31a)
```

Note that the mapping array increases the storage between 25% and 100%, depending on the word sizes of symbolic and numeric data.

3) Duplicate positions are often created in the determination of array NA, as when several numeric values must be added to produce a single matrix position. This duplication results from the creation of a matrix position associated with several physical components. As illustrated in Figure 12, this situation is readily handled with the MAP array, where MAP(2) = MAP(4) = 3.

3. Scalar case: algorithm

In addition to the storage introduced by the MAP array, another critical question is the computational complexity of the MAP generation. This proceeds immediately from a study of the algorithm.

Let a user-supplied n x n matrix structural and numerical description be described by

$$\text{(row,column) position} = (i_k, j_k) \qquad k=1,2,\ldots m$$

$$b_k = a_{i_k, j_k}$$

To arrange in column order, define

$$\ell_k = i_k + (n+1)(j_k - 1)$$

where duplicate values of ℓ_k are allowed. If the values of ℓ_k are sorted in ascending order and a permutation vector p_k maintained so that

$$\ell_{p_r} \leq \ell_{p_k} \qquad r = 1,2,3,\ldots k-1 \qquad k = 2,\ldots m$$

then the sequence

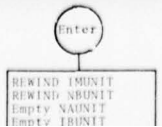
$$b_{p_1}, b_{p_2}, \ldots b_{p_m}$$

is a column-ordered list of numerical values, with possible duplication of positions being adjacent in this list.

Now assume that $\ell_{p_r} = \ell_{p_{r+1}}$ implies that b_{p_r} and $b_{p_{r+1}}$ are to be <u>added</u> to compose a sparse matrix position. Define the set

$$\{q\} = \{r : 1 \leq r \leq m, \ \ell_{p_r} \neq \ell_{p_{r+1}}\} \tag{32}$$

Here, q_k, k=1,2,...s, points to a set of unduplicated values



of ℓ_{p_r}. The column-ordered matrix positions are then given, without duplication, by

$$\hat{j}_k = \lceil \ell_{p_{q_k}} / (n+1) \rceil + 1$$

$$\hat{i}_k = \ell_{p_{q_k}} - (n+1)(\hat{j}_k - 1) \qquad k=1,2,\ldots s$$

The mapping array MAP is generated by defining

$$w_{p_1} = 1$$

$$w_{p_r} = w_{p_{r-1}} \qquad \ell_{p_r} = \ell_{p_{r-1}} \qquad r=2,\ldots m$$

$$= w_{p_{r-1}} + 1 \qquad \ell_{p_r} \neq \ell_{p_{r-1}} \qquad (33)$$

Then the packed sparse array A is calculated columnwise from

$$c_{w_k} \leftarrow c_{w_k} + b_k \qquad k=1,2,\ldots m$$

Thus the set w_k performs the function of the MAP array.

The complexity of the above computations is

(1) $0(m\log_2 m)$ for performing the sort, and

(2) $0(m)$ for performing the scans of (32) and (33).

The sort easily dominates the computation. Since $m=0(n)$, for finite element grids, this complexity is $0(n\log n)$.

4. Vector case: introduction

Large sparse equations are usually most easily formulated in blocks, each relating clusters of equations and variables. These blocked submatrices must then be arranged in column order within the block, and then these columnwise representations inserted into the overall column-ordered matrix structure. This unblocking process - the primary function of subroutine UNBLOK - is complicated by the following factors.

1) Blocks may be generated in any order.

2) Blocks may in general overlap in either or both dimensions (however, see [20] for methods of eliminating overlap at a cost in matrix size).

3) Blocks may have an internal sparsity structure worth preserving, e.g.,diagonal, tridiagonal, etc.

4) For very large matrices, the blocks may not be containable in local store, but must occasionally be written to backing store.

To reduce user concern for these issues, the preprocessing subroutine UNBLOK accepts a high level description of the block structure, and, similar to the scalar case of Figure 12, produces both (a) a vectorized matrix symbolic description for VMSP, and (b) a vectorized mapping array to assist equation formulation.

Since the primary use of the blocking feature is expected to be in the solution of finite element problems, a simplified format has been provided in this case. Entering only the node numbers of the element results in the necessary vectorized matrix description and formulation map, assuming all nodes in the finite element are coupled in the element matrix description. Boundary conditions are also handled at this level.

A summary of scalar and vector formats acceptable to UNBLOK is given in Table 13.

Unpartitioned form

Temporarily ignoring the problem of incorporating a backing store when the matrix cannot reside in local store, two examples will be studied to illustrate the correspondence between symbolic and numeric arrays and their use.

The first example of Figure 13 illustrates the processing of a variety of overlapping block structures. The user supplies a symbolic array NA to the symbolic preprocessor UNBLOK and a numeric array to the equation solver. The latter array must be column ordered by the user within each block, regardless of the block structure. Thus, a finite element must be formed in column order, usually a minor restriction.

Structural Description	Format
Scalar in (i,j) position	-j -i
Dense vector in column j from row i_1 through row i_2	-j i_1 i_2
Finite element connecting nodes $k_1, k_2, \ldots k_{m_1}$ and boundary conditions $\ell_1, \ell_2, \ldots \ell_{m_2}$	k_1 k_2 ⋮ k_{m_1} ℓ_1 ⋮ ℓ_{m_2}
Dense (q=1) Diagonal (q=2) Tridiagonal (q=3) } block between positions $(i_1,j_1,)$ and (i_2,j_2)	0 q i_1 j_1 i_2 j_2

Table 13. Format of NA array input to UNBLOK, assuming column-ordering (using VMBPC, ZMBPC).

The formulation process may be vectorized with a potential savings in storage. Rather than generating a single

array MAP, one can produce both (a) MAP(I) which points to the beginning of a column-ordered vector in the packed matrix array A, and (b) LEN(I) which gives the length of this vector. The A array is then loaded from the B array in this manner.

```
      N=0
      DO 1 J=1,KT
      IV1=MAP(J)
      IV2=LEN(J)+IV1-1
      DO 1 I=IV1,IV2
      N=N+1
    1 A(I)=A(I)+B(N)                                    (34)
```

As in the scalar case, (Eq. (31a)), the array B need not be formed in its entirety before transfer to A. For example, B need store only the numerical components of a single block or finite element; so long as the J and N counters of (34) are maintained, the innerloop can be entered at any time.

The choice of using scalar or vector representation of (31a) or (34) depends on the average vector length (L_{ave}) of the components of the B array. Since this L_{ave} will be less than the L_{ave} for the completed matrix which in turn will be less than the L_{ave} of triangular factors L and U due to fill, the scalar mapping procedure of (31a) is expected to be preferred in the majority of cases.

The second example of Figure 14 illustrates (a) the simplicity of using the high level finite element feature, and (b) a simplified method of incorporating boundary conditions. In this example, all elements are assumed triangular with two unknowns/node. The related matrix structure is shown by x's,

$$\begin{bmatrix} 5. & 2. & 0. & 3. \\ 8. & 6. & 0. & 7. \\ 0. & 0. & 0. & 0. \\ 7. & -2. & 0. & 6. \end{bmatrix} + \begin{bmatrix} 1. & 2. & 0. & 2. \\ 6. & 3. & 5. & 3. \\ 0. & 7. & 4. & 4. \\ 0. & 0. & 0. & 1. \end{bmatrix} + \begin{bmatrix} 3. & 5. & 8. & 5. \\ 7. & -1. & 4. & 0. \\ 3. & 0. & 3. & 8. \\ 0. & 4. & 0. & 0. \end{bmatrix}$$

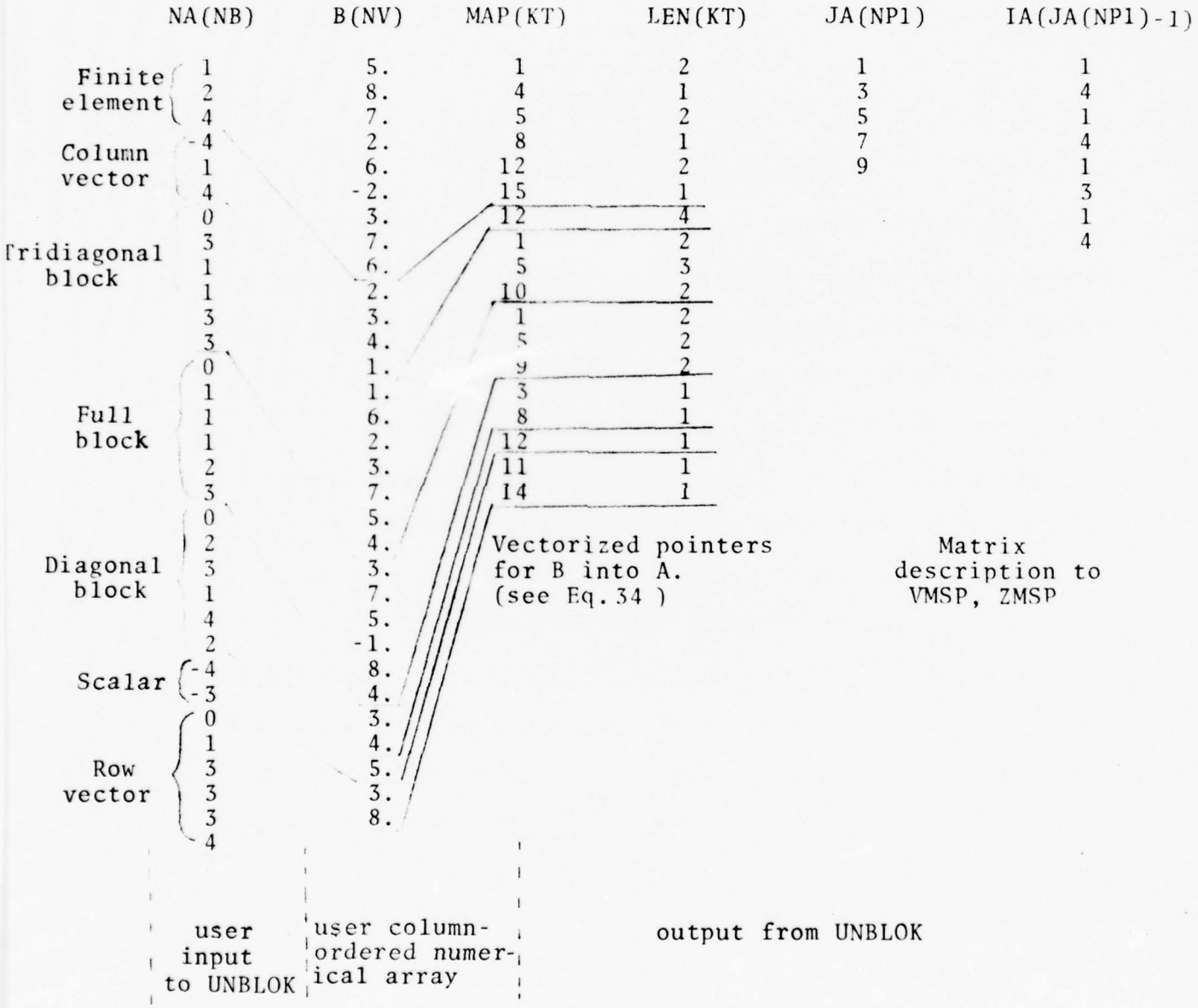
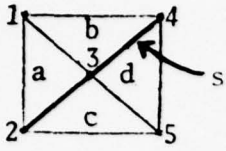
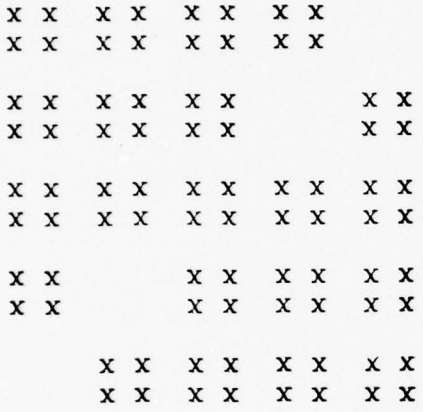


Figure 13. Example of column-ordered matrix preprocessing of structural data.



separator for three partitions

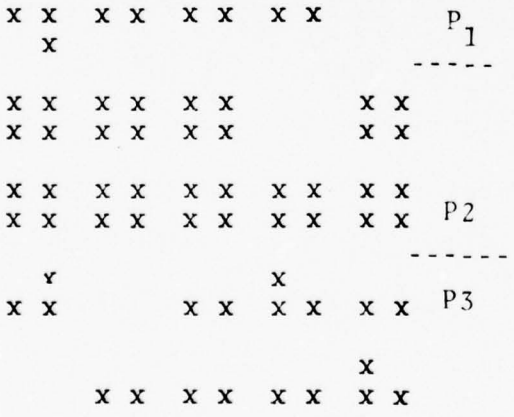
(a) Triangular finite element structure



NA: 1 2 3 1 3 4 2 3 5 3 4 5

(b) Matrix and NA array without boundary conditions

NA: 1 2 . . . see above . . . 4 5 2 7 9



(c) Partitioned matrix and NA array with boundary conditions on variables 2, 7, 9.

Figure 14. Example of row-ordered partitioned finite element problem and matrix.

and the NA array is also given. If the boundary conditions are of the Dirichlet type on variables 2,7, and 9, these numbers can simply be appended to the NA array as shown. If row-ordering is used (VMBPR), subroutine UNBLOK will then generate appropriate vector descriptors to numerically zero elements in the rows associated with the boundary condition variables. A unit-valued diagonal will then force the variable to the right hand side value. An example will be given shortly.

Partitioned form

When the A array cannot be kept in local store, it must be formed in partitions. If the matrix is row-ordered as required for incorporation of boundary conditions, the partitions must also be along row boundaries, as illustrated by the three partitions (P1-P2-P3) of Figure 14.

Each partition is formed in a buffer region of local store; when a partition is completely formed, it is written out to backing store, and the buffer region is overwritten by the next partition. Continuing with the example, assume that each finite element is formulated before beginning the next. Then the coupling of elements sharing nodes within the element array requires that at least one partition be resident in local store, representing the coupling of past and future partitions. Thus, P1 representing node 1 of Figure 14 cannot be written to backing store until finite elements a and b have been formed. Then the separator 2-3-4 isolates succeeding elements from node 1, so that P1 can be written to free space for P3.

A subtlety in this buffering scheme is introduced by distinguishing the extent of concurrency by the processor in

the formulation and I/0 processing. If the central processor performs the I/0 as well as the numerical equation formulation, then only two buffer regions need be defined. For example, in the 5-partition problem of Figure 19, to be discussed in detail later, the formulation-I/0 sequence with two buffers would proceed as follows.

Set up elements a_1 - a_8 in buffers 1 and 2
Write buffer 1 (cols. 1-10)
Set up elements b_1 - b_8 in buffers 2 and 1
Write buffer 2 (cols. 11-20)
Set up elements c_1 - c_8 in buffers 1 and 2
Write buffer 1 (cols. 21-30)
Set up elements d_1 - d_8 in buffers 2 and 1
Write buffers 2 and 1 (col. 31-46)

On the other hand, if the I/0 is handled by a separate processor and a buffer region cannot be simultaneously filled and drained, three buffers must be used. The example would now proceed as follows.

Set up elements a_1 - a_8 in buffers 1 and 2
Set up elements b_1 - b_8 in buffers 2 and 3; write buffer 1
Set up elements c_1 - c_8 in buffers 3 and 1; write buffer 2
Set up elements d_1 - d_8 in buffers 1 and 2; write buffer 3
Write buffers 1 and 2.

5. Vector Case: algorithm

The subroutine UNBLOK operates on user-supplied block descriptions as follows. Since the blocks produced in the vector case have an internal sparsity structure, the first step in column-ordering the entire matrix is to column-order each block. This is carried out "on the fly", i.e., as

each block is recognized in the user description, an expanded vector description - beginning and ending row numbers of each dense column segment - is generated in column order for the block. In the case of a finite element, the entire submatrix is column-ordered, irrespective of the block structure.

The arrangement of these vectors into a column-order for the entire matrix has two steps.

(1) An array of beginning row numbers for each vector is saved from the above process. Together with the corresponding column number, these "scalar" descriptions of each vector starting position are sorted as described in the scalar case.

(2) The column-ordered list of starting vector positions is scanned a column at a time. For each column, starting and ending row positions of vectors are noted, and overlapping vectors are combined to form the final compacted vectorized matrix structure.

The sorting is again the dominant computational effort. If there are m_1 blocks with an average of m_2 columns/block, then the complexity is $0(m_1 m_2 \log_2(m_1 m_2))$.

D. Program Flow Charts

Having illustrated the use of the unpartitioned solver VEGES and the symbolic preprocessor UNBLOK, we may now view the flow chart of the complete system package with some understanding before considering in detail the more complicated partitioned version in VEGES/P. Particularly noteworthy in Figure 15 is the IBM 360/370 assembly language version of the

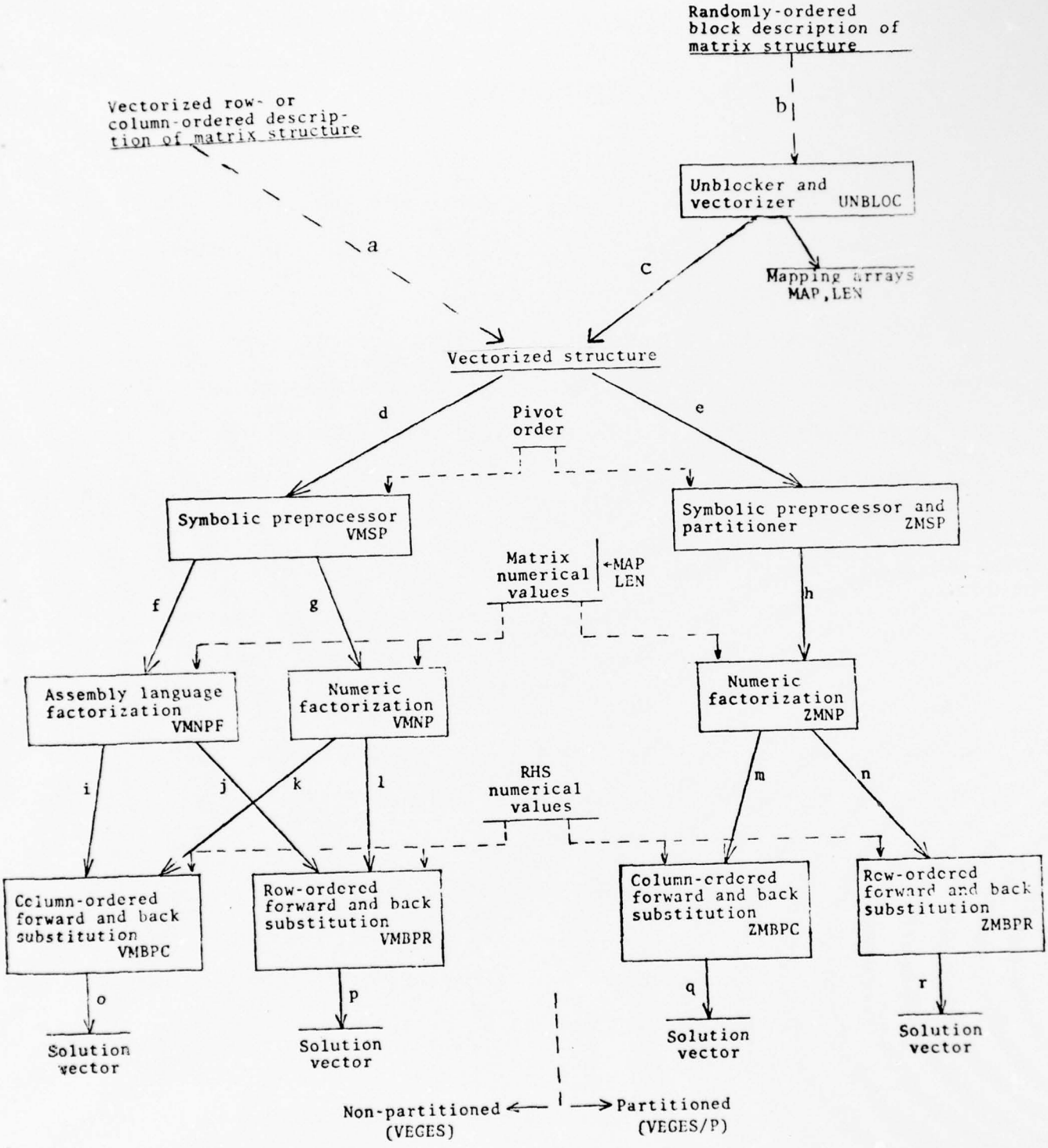


Figure 15. Flow-chart showing options in dual sparsity package; All blocks coded in Fortran unless otherwise noted; see Table 16 for complete list of subroutines associated with each block.

2). The key steps in this process are the following.

numeric factorization portion of VEGES.

E. Details of the Partitioned Solution

1. Introduction

The implementation of a backing store version of a general equation solver necessitates a fundamental assumption concerning the size of local store versus the size of the system of equations. In contrast to specialized full-, band-, and block-solvers where the structure is given, sparse solvers must have ready access to both numerical and structural data; this implies that a decision must be made on whether to maintain either or both completely in local store or whether only a local description - adequate for a local computational sequence - should be maintained, the rest residing on backing store. A similar consideration is involved in the equation formulation stage, i.e., whether the matrix (and the MAP array if UNBLOK is used) is formulated and written to backing store in parts. In general, a tradeoff must be made between flexibility in use when all potentially large arrays are partitionable, and user convenience when all arrays are resident in local store.

In VEGES/P, it is assumed that arrays associated with the symbolic solution phase are in local store but all other large arrays are in backing store. Table 14 details this assumption by giving the residency status of A, L, U, etc. in the formulation and solution stages. The justification for this choice is that the vectorized structural arrays are likely to require

at least an order of magnitude less storage than numeric arrays

array	residency
A	
structural	L
numerical	B
MAP, LEN	B

(a) Formulation

	residency	
array	numerical	structural
A	B	L
B	B	-
L,U	B	B *
X	L	-

*L during symbolic phase

(b) Solution

Table 14. Location of arrays in UNBLOK and VEGES/P; L - local store B - backing store.

for problems necessitating a backing store (the ratio being 4n for L and U in the finite element problem class of Appendix table A1).

2. Flow chart of UNBLOK symbolic preprocessor

The UNBLOK subroutine originally cited in Figure 12, has the ability to partition the MAP and LEN arrays, as indicated in Table 14, writing these to a backing store one partition at a time. Thus, these formulational arrays do not seriously affect the storage requirements for the symbolic phase of the solution.

Other noteworthy features of UNBLOK are the ability (a) to produce either scalar or vector mapping arrays (see Eqs. (31a) and (34)), and (b) to identify rows which are to be zeroed in the manner of Figure 14c, to handle boundary conditions.

3. Flow chart of ABLOK numerical preprocessor

The numeric formulation phase is complicated somewhat by partitioning of the formulation step, since the MAP and LEN arrays are on backing store (from UNBLOK above); then retrieval must be coordinated with the formulation of components of A.

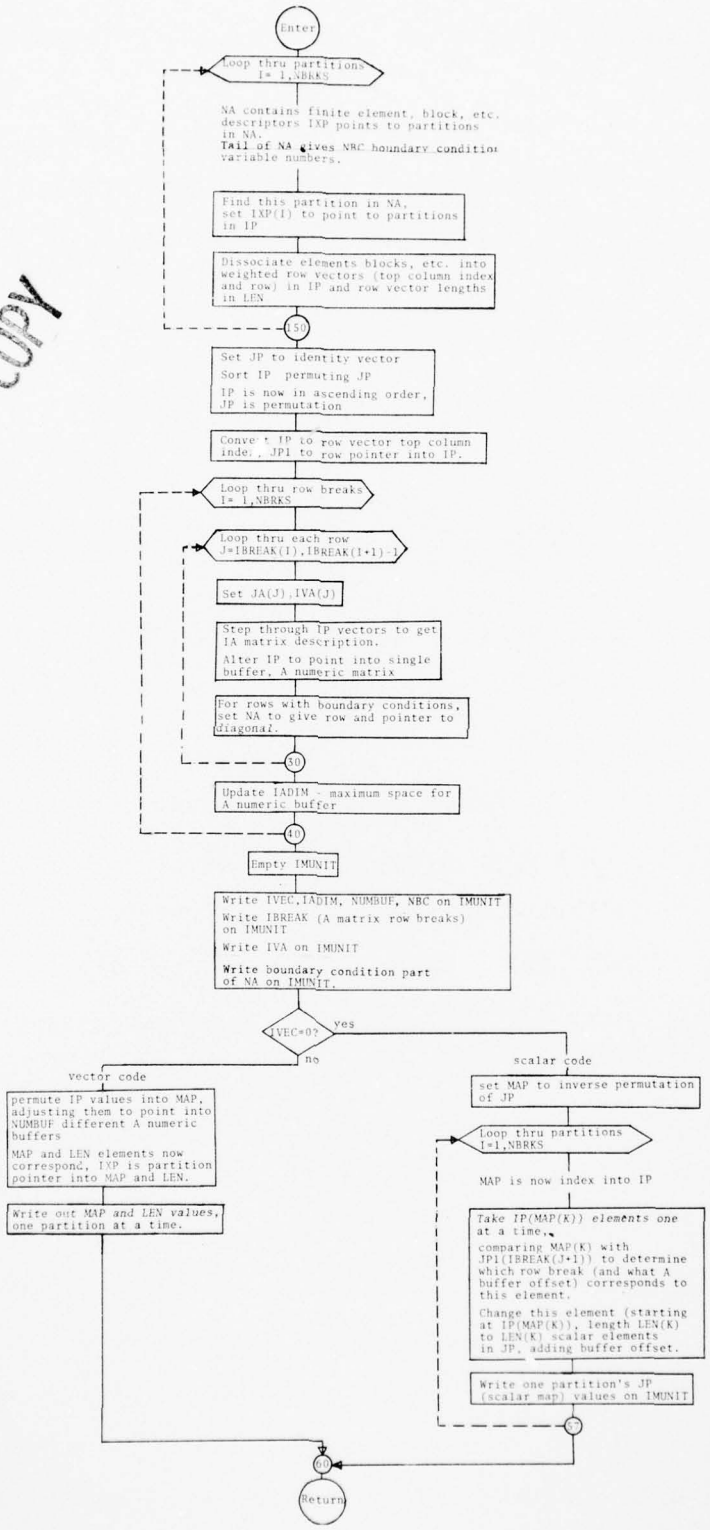


Figure 16. Flow chart of UNBLOK subroutine

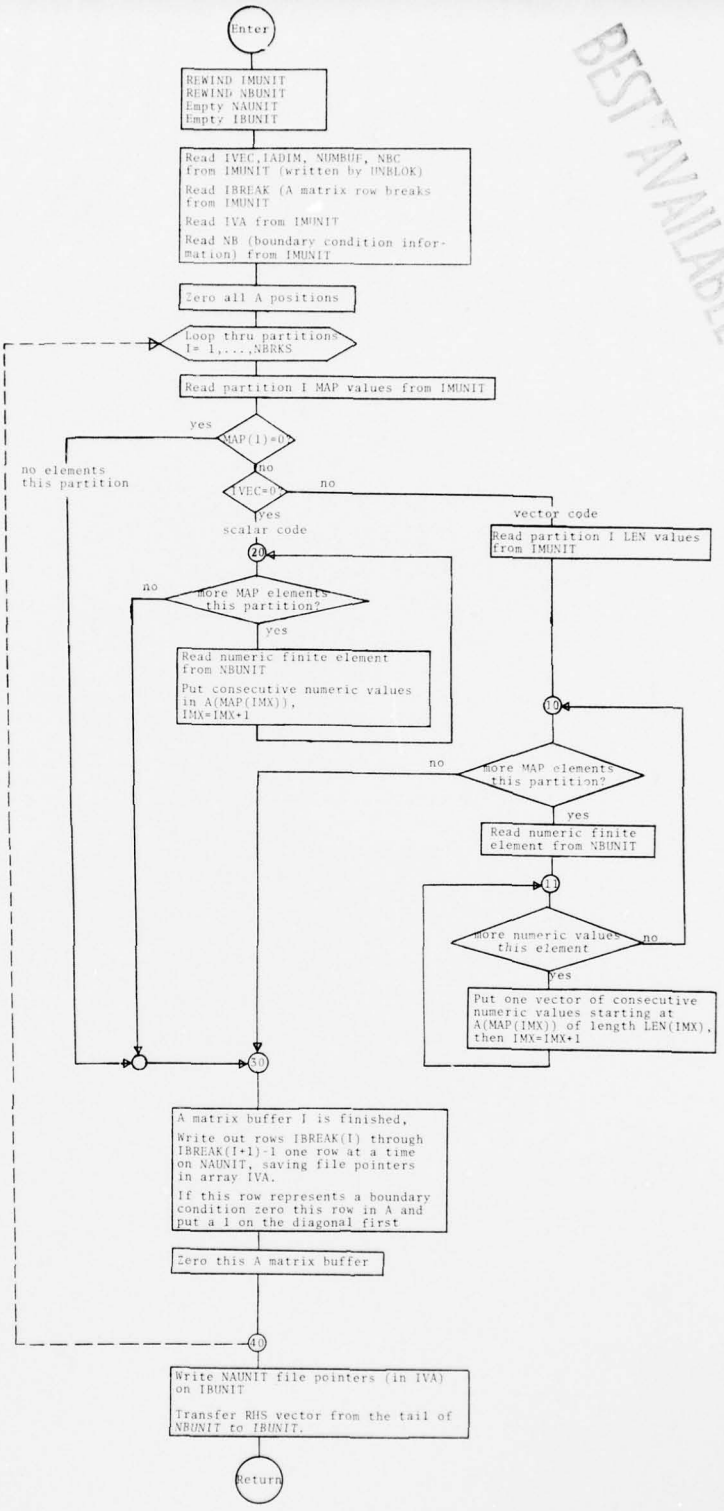


Figure 17. Flow chart for subroutine ABLOK

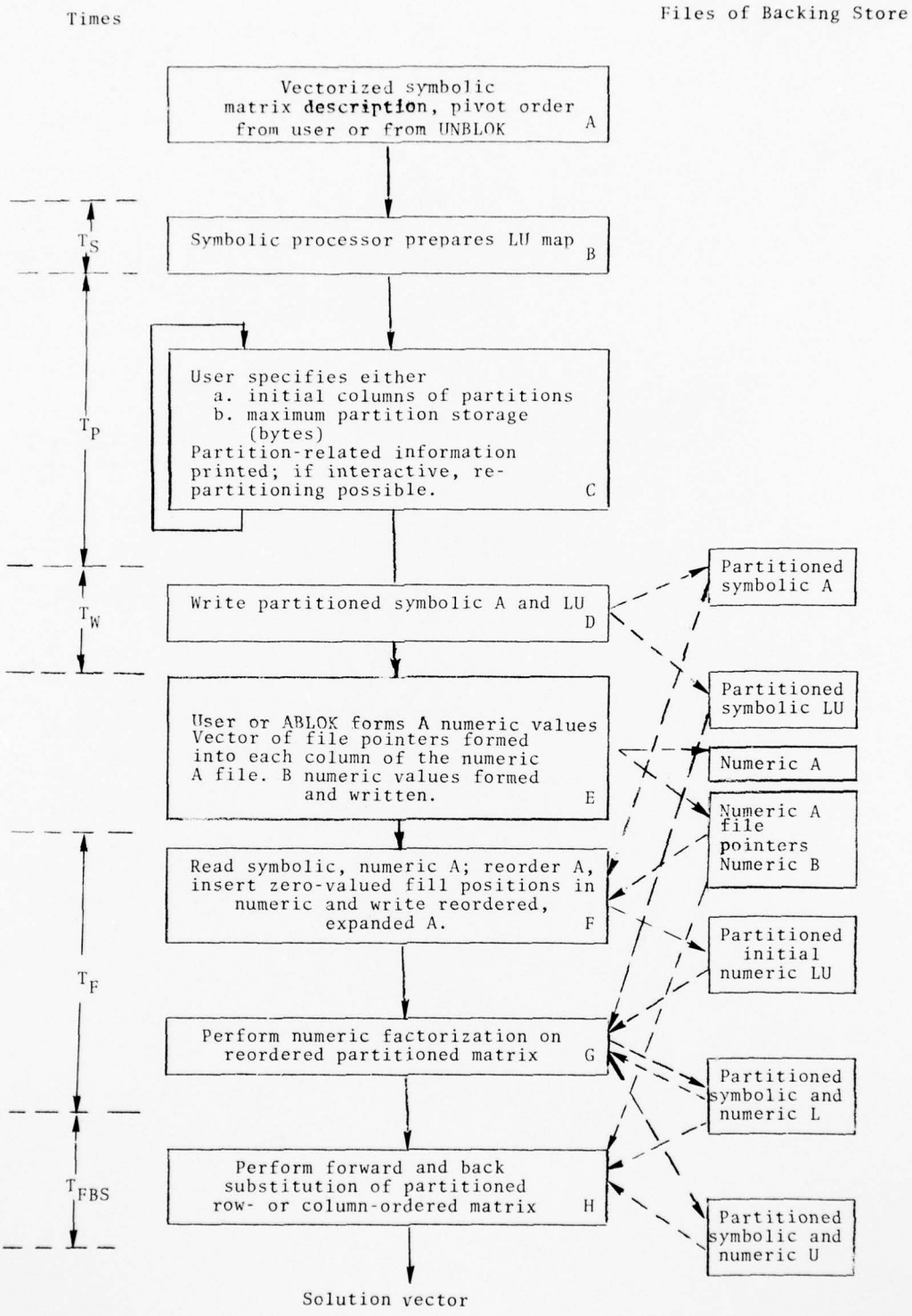


Figure 18. Flow chart of partitioned solution.

This coordination can be generalized provided that the components of A are also on a backing store in the order consistant with MAP and LEN. This is performed by subroutine ABLOK with flow chart given in Figure 17. Note that use of this routine precludes formulation of the matrix "on-the-fly" from its components, and requires an additional read/write cycle in the equation formulation. This price for generality is expected to be small for fast scalar processors but could be significant for vector processors. At a minimum, ABLOK provides an example of the coordination process for the user with an alternate strategy.

4. Flow chart of symbolic, numeric solution

The symbolic and numeric formulational preprocessors leave the matrix structure and values on a backing store, respectively. The symbolic and numeric solution then proceeds in five major steps.

(1) The columns which initiate the partitions are selected. These column breaks are determined in an addition to the symbolic processing phase by user specification of either (a) column numbers, thus accepting whatever partition storage requirements that result, or (b) maximum strip storage size, permitting the column breaks to be selected by an internal algorithm. This interactive partitioning step is depicted in the block diagram of Figure 18 and will be illustrated shortly.

(2) The matrix A is read, reordered, and written to backing store with zero-valued positions of L and U inserted.

(3) The L and U are formed in column-ordered strips, each strip being written to backing store on completion and recalled when necessary to form another column strip.

(4) The strips of first L and then U are recalled in sequence to carry out the forward and back substitution steps. These steps (F-H) are shown in the flow chart of Figure 16, together with the specific reads and writes to backing store of both numeric and symbolic information (the times given are referenced in Appendix table A4).

5. Example of partitioned solution of finite element problem

The finite element grid of Figure 19 presents a sufficiently complicated problem to illustrate the major features of VEGES/P and the relative simplicity of solving this large class of application problems.

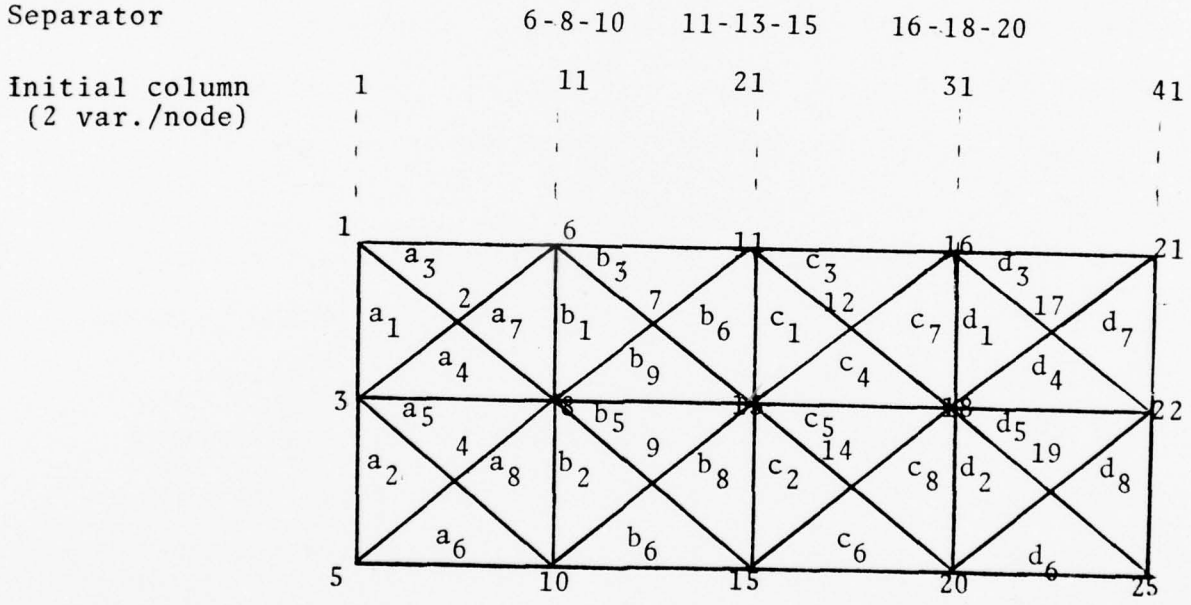


Figure 19. Finite element array and partitions for UNBLOK.

The symbolic data presented to UNBLOK consists of principally the finite element node numbers, the boundary variable numbers, and partitioning information for both the variable numbers and the finite element node number list. The precise data requirement is shown in Table 15. Note that these

```
N, the size of the matrix
IVEC - 0 for scalar formulation
     - 1 for vector formulation
NUMBUF, the number of buffers
NPART, the number of partitions
NBC, the number of boundary conditions
NVAR, the number of variables/node
NNODE, the number of nodes/element
       both in COMMON/FEL/
NA(NNODE*NVAR*(no. of elements)+NBC),
       list of element node numbers and
       boundary conditions
IPART(NPART+1), beginning column (row)
       numbers of each partition
IXP(NPART+1), pointer into NA indicating
       beginning of new partition
```

Table 15. List of input data to UNBLOK.

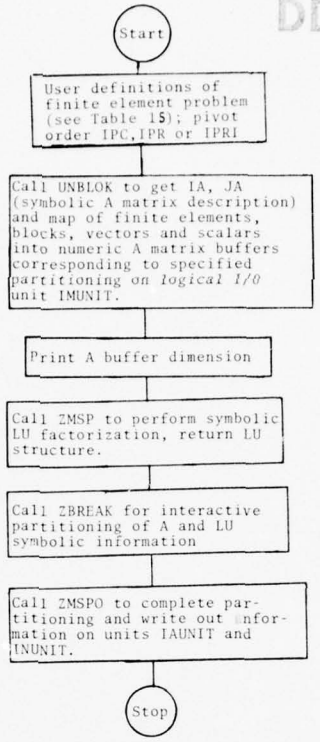
partitions pertain only to the equation formulation step. In this example, this data is furnished by DATA statements.

As the flow chart for the example shows (Figure 20), the symbolic phase can be executed through ZMSP (the preparation of the LU map) with this minimal information. The partitioning of the matrix solution itself is next carried out interactively. The interaction is illustrated in Figure 21, where two partitions are examined - one based on a maximum buffer size, and one on specific column breaks (which happens to be identical to the

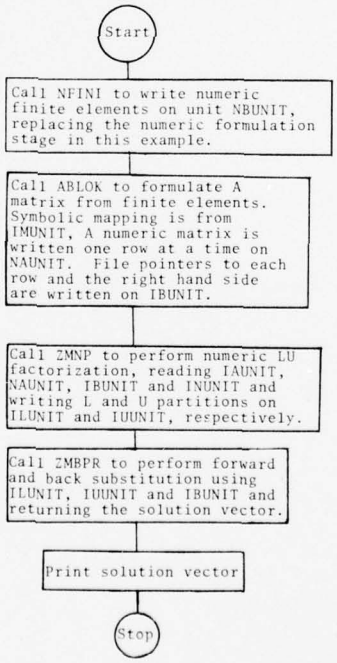
column breaks specified in the formulation stage in this case). The choice of a partition results in the writing of the L and U maps as well as the presentation of certain critical dimensioning information for the numeric solution phase.

The finite element is evaluated and written to backing store in the same order as presented in array NA. These are retrieved together with the MAP and LEN arrays in ABLOK, which then writes the partitioned A matrix to a backing store. The remainder of the solution follows in the manner of Figure 18.

The main programs for the symbolic and numeric solution phases are given in Appendix C. The reader will note that in the numeric phase the user need supply only the subroutine NFINI for evaluating the finite element matrix and the right hand side vector.



(a) Symbolic Main Program - includes all dimensioning for calls to UNBLOK, ZMSP, ZBREAK and ZMSPO.



(b) Numeric Main Program - includes all dimensioning for calls to NFINI, ABLOK, ZMNP and ZMBPR.

Figure 20. Flow charts of example finite element solution program of Appendix C.

```
#run loads+zsymf+zlib+vmlib 0=-im 1=-ia 2=-in 7=#sink# t=2
#EXECUTION BEGINS

 DIMENSION A(  248)

 ZBREAK: ENTER MAX ARRAY STORAGE (BYTES)
 2464,


   BLOCK   LOW COLUMN   L STRT   MIN L SIZE   MIN IXBUFF
      1           1        1         468         1224
      2          13        1         376         1192
      3          22        1         372         1192
      4          31        2         392         1184
      5          41        4          96          776

   BUFFER SIZES - ILENSY UNITS
 IBBUFF       852
 IXBUFF      1224
 A             72
 ZMBF         468

   I/O - BYTES
 WRITE      12280
 READ       17578

 ZBREAK: OK ? (EOF=NO)
 \

 ZBREAK: ENTER NUMBER OF BLOCKS
 5,

 ZBREAK: ENTER STARTING COLUMNS FOR BLOCKS, 10 PER LINE
 1,11,21,31,41,


   BLOCK   LOW COLUMN   L STRT   MIN L SIZE   MIN IXBUFF
      1           1        1         384          960
      2          11        1         416         1232
      3          21        2         416         1232
      4          31        3         392         1208
      5          41        4          96          776

   BUFFER SIZES - ILENSY UNITS
 IBBUFF       892
 IXBUFF      1232
 A             72
 ZMBF         416

   I/O - BYTES
 WRITE      12272
 READ       15890

 ZBREAK: OK ? (EOF=NO)

  SUGGESTED EQUIVALENCING:
  (A,IXBUFF)
  (IBBUFF,IXBUFF(      73))
#EXECUTION TERMINATED
#run loads+znum+zlib 0=-im 1=-ia 2=-in 3=-na 4=-ib 5=-nb 8=-ja 9=-il 10=-iu t=2
#EXECUTION BEGINS

 SOLN VEC
    1.0000         1.0000         2.0305         2.0305         1.0000
    3.1254         3.5682         3.5682         1.0000         4.0754
  -0.10502E-01   -0.10502E-01     -2.2140        -2.2140         0.75874
   0.75874        -1.4452        -1.4452         1.5272         1.5272
   -2.6924        -2.6924        -1.4452        -1.4452        -2.7339
   -2.7339        -2.2140        -2.2140        -2.6924        -2.6924
    1.5272         1.5272         3.5682         3.5682         0.75874
   0.75874         2.0305         2.0305      -0.10502E-01   -0.10502E-01
    1.0000         4.0754         1.0000         3.1254         1.0000
    1.0000
#EXECUTION TERMINATED
#
```

Value returned from UNBLOK for A numeric buffer for ABLOK.

Symbolic partitioning routine ZBREAK permits examination of partitioning strategies and their effects on numeric buffer storage and I/O requirements.

Partitioning can be specified by either of two methods:
(1) by entering a maximum array storage figure which determines column breaks by limiting the total buffer storage to the given number (this does not include other required arrays of length N).
(2) by entering specific column breaks.
The choice is controlled by end-of-file entry by user

L STRT* gives number of first L block needed in LU factorization of this block.

MIN L SIZE is the size in "ILENSY units" (IBM 360,370, AMDAHL 470V/6: halfwords) of L buffer containing symbolic and numeric information. Since previous L blocks are needed for factorization this number is critical to the amount of I/O done in the numeric routines.

MIN IXBUFF is the size in ILENSY units of the major symbolic and numeric buffer required for this partition.

BUFFER SIZES are presented for dimensioning purposes in the numeric main program; I/O gives a count of all read and write operations.

*ZBREAK headers consider the matrix to be column ordered; for row ordering exchange "U" for "L" and "row" for "column."

Equivalencing is suggested on the basis that the A and IBBUFF arrays must be kept separate but can overlap IXBUFF.

Numeric execution is non-interactive, the solution vector being returned from the (in this case row ordered) back substitution routine.

Figure 21. Run of symbolic and numeric program of Appendix C.

F. Fortran Implemetation (VEGES)

1. Symbolic processing

A Fortran implementation of these vector methods can be viewed in three distinct steps (see Fig.11). Given the symbolic map of A in IA and JA, the symbolic processing routine VMSP identifies all fill positions and creates maps of the L and U matrix factors. Array results IU and JU from VMSP form a vectorized U symbolic map. IVU(J) is an index pointing to the start of the J'th column of numeric elements in the (yet to be calculated) packed U numeric array. Similarly, IVA points to the start of each column in the packed A numeric array. IL, JL, and IVL describe the L matrix. We shall examine how these arrays are determined.

The method for finding symbolic fill loops through each column. Counters are kept and updated for symbolic and numeric positions in L and U. These are used for JU, JL, IVU, and IVL. The permuted A column is converted to scalar row indices, row permuted, and converted back into vector form. The code then loops through each U position, examining the corresponding L column and updating or inserting vectors in the current column to account for fill locations. When there are no more U positions, the vector representation of this column is a map of LU for that column. It is split at the diagonal and copied to IU and IL. The symbolic processing of this column is now complete.

2. Numeric factorization

The A numeric array and all symbolic information is input to the numeric factorization routine VMNP. Since column ordering is

VEGES subroutines	
Symbolic	VMSP
	VSORT
Numeric	
Factorization (Fortran)	VMNP
Fact. (Fortran - Assembly, column-ordered)	VMNPF
	VMNPA
Column-ordered substitutions	VMBPC
Row-ordered substitutions	VMBPR
VEGES/P subroutines	
Symbolic	ZMSP
	VSORT
	ZBREAK
	ZMSPO
Numeric	
Factorization	ZMNP
	ZMNPI
	ZMNPA
	ZMNPB
	ZMNPO
Column-ordered substitutions	ZMBPC
Row-ordered substitutions	ZMBPR
I/O routines	ZLIB
Symbolic and numeric preprocessing into column-ordered vectorized lists from randomly-ordered (i,j), block,or finite element lists	UNBLOK
	VSORT4
	ABLOK

Table 16. Subroutine lists for factorization

used, standard LU factorization is carried out using Eq. (12). Returned are packed numeric arrays L and U corresponding to IVL and IVU, respectively. Also returned is DI, an array of the inverse pivot elements.

Looping through each column , the symbolic bounds for this column in $\underline{U}$ and $\underline{L}$ are picked up from JU and JL. These point to this column in IU and IL. The bounds of this (permuted) column in $\underline{A}$ are retrieved. Calculations are done on a full column length (unpacked) numeric array illustrated in Figure 4a. Row permuted A values are copied into this (initially zero) scratch array. We now loop through each $\underline{U}$ position in this column. The numeric multiplier is copied into U and bounds on the corresponding $\underline{L}$ column are examined. Executing as a vector instruction, the packed L column times the numeric multiplier is put in another scratch array T*. Looping through each L vector, values from T are subtracted from the current X column as in Figure 3 . After all $\underline{U}$ elements in X have been considered, the inverse of the diagonal is stored in DI. Remaining $\underline{L}$ vectors in X are multiplied by the inverse diagonal and stored in packed form in L. This process is executed for each column of the matrix.

3. Forward and back substitution

This stage of the solution solves the two systems $\underline{L}\ \underline{y} = \underline{b}$ and $\underline{U}\ \underline{x} = \underline{y}$. Given numeric and symbolic $\underline{L}$ and $\underline{U}$ arrays and DI and B, the right hand side, VMBPC employs Equations 13 and 14 to solve the

*This describes the option when separated multiply and subtract instructions are used.

AD-A037 971 MICHIGAN UNIV ANN ARBOR SYSTEMS ENGINEERING LAB F/G 12/1
VECTORIZED GENERAL SPARSITY ALGORITHMS WITH BACKING STORE.(U)
JAN 77 D A CALAHAN, P G BUNING, W N JOY AF-AFOSR-2812-75

UNCLASSIFIED SEL-96 AFOSR-TR-77-0259 NL

2 OF 2
AD A037971

END
DATE FILMED
4-77

forward and back substitution. The solution vector $\underline{x}$ is returned in B after the appropriate permutations.

The right hand side is first row permuted into a scratch vector X. Looping through all but the last column in the forward substitution, for column J, X(J) times the packed L column is stored in the temporary array T in a vector instruction. This packed T array is subtracted from the X vector, again as in Figure 3. Completing the forward substitution, the X vector now contains $\underline{y}$. The back substitution proceeds similarly from the last column through the second column, though X(J) is first multiplied by the inverse diagonal. The solution in X is last transferred to B using the column permutation.

4. Conclusions

Figure 11 is a flow diagram for this non-partitioned program, showing input and resultant arrays. The symbolic processing is executed first and is independent of any numeric arrays. The result of this is an $\underline{L}$ $\underline{U}$ map which can be used for any matrix of this size and structure. Next the numeric factorization and forward and back substitution are carried out on the $\underline{A}$ matrix and right hand side. Note that for different $\underline{A}$ numerical values only VMNP and VMBPC must be repeated, and, when B changes, only VMBPC need be repeated. If the matrix is originally given in row order , VMBPR is used in place of VMBPC.

G. Fortran Implementation (VEGES/P)

1. Symbolic processing

Symbolic processing in VEGES/P does not consider the matrix

partitioned, assuming that all symbolic descriptors can all fit into local stores. This assumption aids the partitioning process considerably, although possibly restricting the size of problems which may be solved.

The symbolic part of VEGES/P does not produce separate pointer arrays into L and U, so that vectors may cross the diagonal. This aids numeric computation in the factorization routines, as fill vectors are not broken unnecessarily. Symbolic vectors are described in IX, a combination of IU and IL in VEGES. JU and JL point into IX, with JL pointing to the first vector that runs into L. IVU and IVL point into separate L and U arrays. JL and IVL will not be passed to the numeric routines because the diagonal can be easily recognized by comparing the row index with the column number while stepping through U vectors, thus saving the I/O necessary to transfer these arrays. Besides these symbolic descriptors, arrays LA and KA are passed to the numeric routines; these correspond to JA and IA except that row and column permutations have been applied.

2. Partitioning

Results from symbolic processor ZMSP are passed to an interactive routine ZBREAK, which, on the basis of an entered maximum variable array storage, calculates where the matrix should be partitioned. The corresponding buffer size for each of these strips is displayed, as well as the number of the first recalled L strip needed for factorization of each current strip. Maximum overall buffer sizes are printed for use in accurate dimensioning of a factorization and/or substitution driver program. An alternative to this automatic partitioning is also available; here the number of the first column of each strip is entered. The total

number of bytes read and written in ZMNP is also presented, giving an indication of the effect of breaking on I/O operations.

Once a set of partitions is decided upon, ZMSPO is called to write out the two symbolic data files, IAUNIT and INUNIT. IAUNIT contains information needed to reformat the A matrix into numeric buffers in ZMNPI. These arrays are JA, IA, LA, KA, and KVA. Each buffer contains the section of these arrays pertaining to that strip. Note that strips are in the permuted matrix. JA and LA are adjusted to point into the buffer locations of IA and KA, respectively. Vector KVA is created here and points into the X numeric buffer (combined L and U strips) to be written on JAUNIT, indicating the start of every permuted A vector so that, when the buffer is initialized, values can be easily transferred. INUNIT is read by ZMNP and provides the symbolic description for each X block. First, however, permutation vectors IPC and IPRI are written out for use in ZMNPI and ZMBPC. To describe the X block, arrays IVU, JU, and IX are broken and written out. IVU is adjusted to point into the X numeric buffer and JU to point into IX in the buffer. Header information is written out with each buffer, including buffer number, column range, and first L strip for the numeric factorization and substitution routines.

3. Strip numeric factorization

The numeric factorization process is broken up into several steps (Fig 23). Subroutine ZMNPI is called once, initializing the X numeric buffers. Permutation vectors are passed to ZMNPI, which reads symbolic information from IAUNIT. The A matrix is input on file NAUNIT in unpermuted order, written one column per logical

record. File pointers to each column are read in from IBUNIT. (These file pointers were picked up when the matrix was being written, and are the responsibility of the user. Also on IBUNIT is the right hand side, to be read by ZMBPC. Note that with this method of storing the A matrix, only one column must be in memory at a time, but other methods are possible, depending on the limitations of the system.) Strip by strip, the matrix is permuted and fill positions are set to zero. The resultant X numeric buffer is copied out to JAUNIT, to be read later in ZMNP. All partitions are processed at this time so that buffer space needed here can be overlapped with space used in the factorization.

4. Numeric factorization

Factorization takes place one strip at a time. ZMNP reads symbolic and numeric information from INUNIT and JAUNIT, respectively, before passing control to ZMNPA. This routine is given the number of the first L strip needed and an array of pointers to the start of L strips written on ILUNIT. The last array position set points to the current strip, or an end-of-file, since the current strip has not yet been factored. (Note that if a file system has no pointing facility, ILUNIT can be rewound and buffers read starting from number one. The price paid is some unnecessary I/O, depending on the matrix structure.) Having read an L buffer, ZMNPA performs the normal numeric computations which involve only these two (X and L) parts of the matrix. L buffers are read and computations performed until an end-of-file (up to the current strip) is encountered on ILUNIT. ZMNPB is called to perform numeric computation within the current X strip and put the inverse diagonal in DI; the result is

a completely factored strip of the L U matrix. In ZMNP numeric computations there are only two differences from VMNP. Instead of operating on an expanded (unpacked) column before packing it into the L and U arrays operations are carried out on the already packed X numeric buffer. The packed buffer option was specified by the preformatting of the X numeric buffers. Symbolic and numeric pointers have been kept for each column pointing into the U part of the X block. As we step through U, these end up pointing to the start of L. These are effectively IVL and JL.

With IVL, JL, IVU, JU, and IX, we have the information to separate the X block into L and U and write these buffers out as ILUNIT and IUUNIT, respectively. This is done in ZMNPO. IX is broken at the diagonal, JL and JU adjusted to point into their respective parts, and likewise the numeric buffer is broken and IVL and IVU adjusted. Buffers, with appropriate header information, are written out and the position of the ILUNIT file is noted for use in ZMNPA. This completes the operation on this strip, and the next symbolic and numeric buffers are read in.

5. Forward and back substitution

ZMBPC is input permutation vectors IPC and IPRI, data files ILUNIT, IUUNIT and IBUNIT, DI and vector B, scratch vector X and space for any buffer on the L and U data files. ZMBPC performs the forward substitution on L buffers, reading them sequentially from ILUNIT. In the back substitution, however, the U buffers must be retrieved in reverse order. This can be accomplished in various

ways, using either IBM Fortran direct I/O, the BACKSPACE statement, or some system routine taking advantage of a particular file system.

The driver program included with the vectorized sparse matrix package creates a sequence of randomly-positioned matrices with randomly-selected pivot orders. It is intended as a test program, checking the solution vector against one generated with the matrix and flagging errors; it also gives an illustration of program flow and subroutine calls.

A similar driver and matrix generator is included with the partitioning package, with the addition that matrix break points are randomly generated, and the routine interpreting these breaks has been made non-interactive for convenience in running many matrices.

The above program can be used as a non-interactive test program, but by setting a flag, the break point generation will be bypassed and full interaction with the breaking routine is possible.

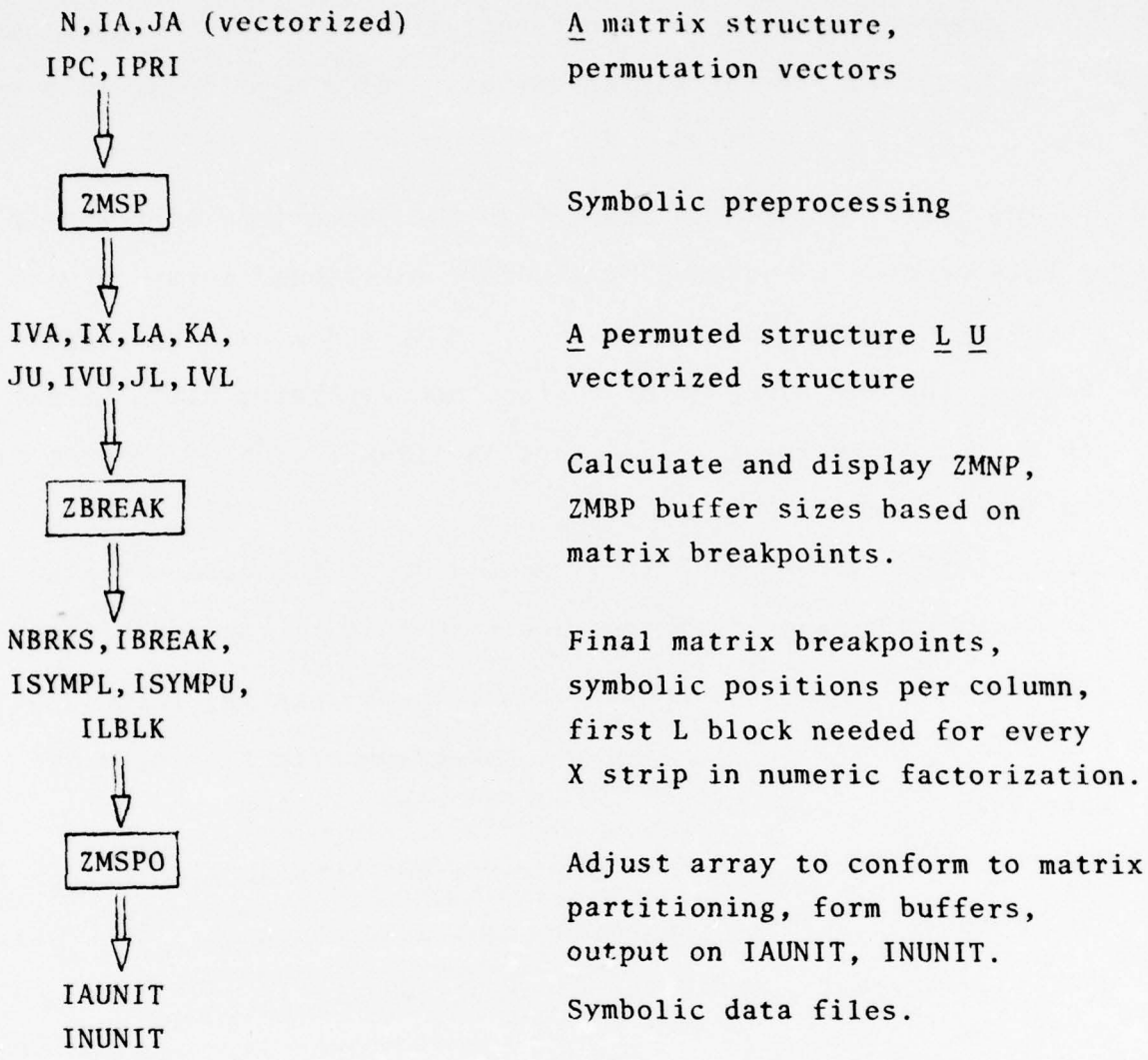


Figure 22. Symbolic factorization flow chart for partitioned matrix solver

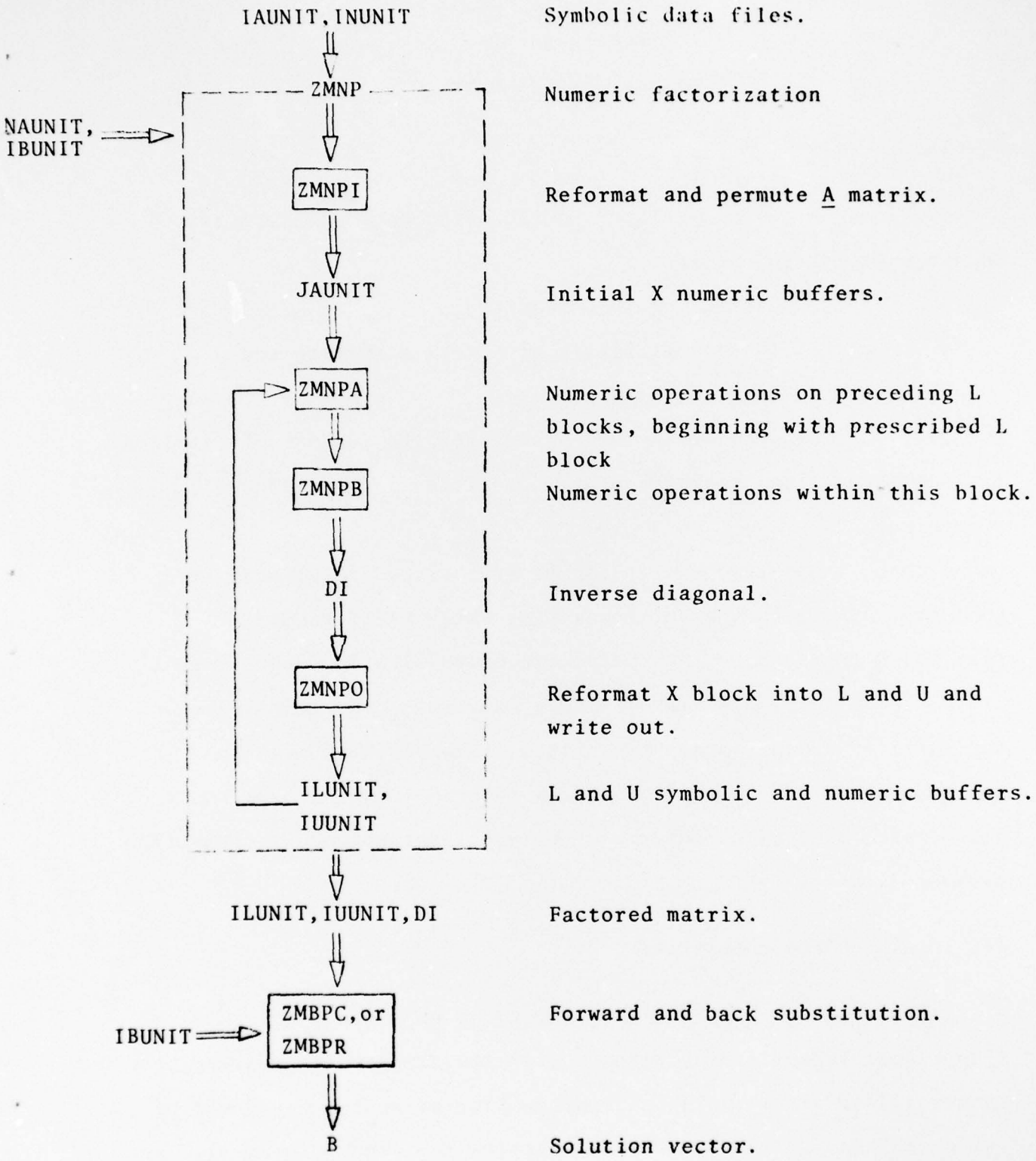


Figure 23. Numeric factorization and substitution flow chart

Appendix A

Numerical Experiments

I. Introduction

Two dominant issues related to the efficiency of a vectorized sparse equation solver are

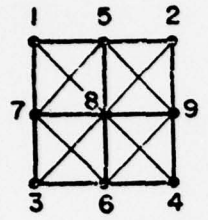
1) matrix size and density,
2) the compatibility of matrix structure and computer architecture.

Therefore, an experimental study should involve a class of matrices of increasing size, with documented densities and structural regularity.

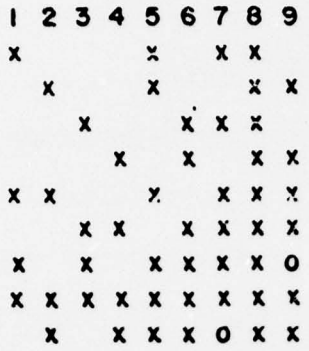
Although these properties can be synthesized relatively independently of one another by generating variants of randomly-positioned matrices, such studies are often viewed as unrepresentarive of commonly-encountered sparse matrices. For this reason, the matrices in this study are either taken directly from an application or synthesized according to grid (mesh) generation rules associated with finite element problems, solved by dissection methods [1].

II. Finite-element Matrices

Illustrated in Figure A1(a), the nodes of a rectangular 2-dimensional linear finite element grid are numbered in a prescribed manner [1] so as to yield local decoupling of rows and columns of the associated matrix, the local coupling replaced by a distributed coupling throughout the matrix. This dissection process proceeds routinely so long as the number of nodes/side is 2^n-1, and is

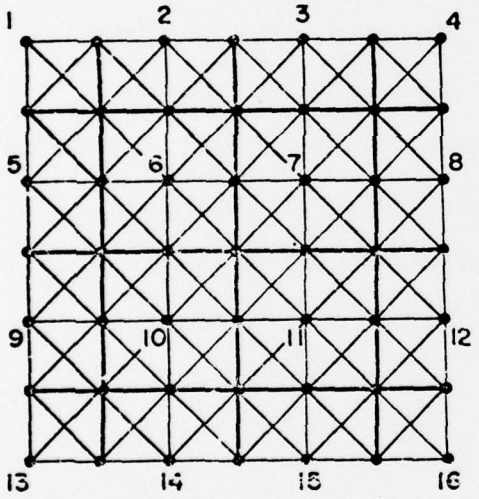


Four element grid and connectivity

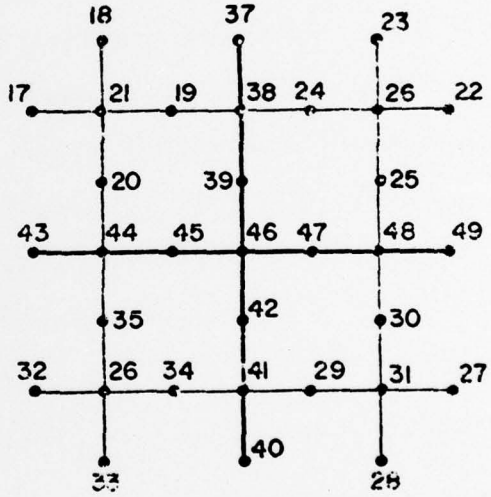


Associated matrix (X) and fill (0)

Figure A1. Simple dissected grid and matrix



Grid connection and initial dissected node numbering



Second (light) and third dissection stages

Figure A2. Larger dissected grid

illustrated for a larger grid in Figure A2.

Due to the regular structure of this finite element family, it is possible to obtain exact arithmetic operation counts and estimates of the vector counts involved in the matrix factorization. The asymptotic components of these formulae (for large grids) are given in Table A1, together with exact counts obtained from experimental solutions of the matrices. This is intended to establish the validity of the formulae used throughout the report to estimate computational complexities of problems beyond the range of the experiments.

Tables A2 and A3 give a variety of structural properties and execution times for n = 2,3,4,5. Most of the results are cited in context throughout the report. Also included are the timings and memory requirements associated with two early procedures [6][9] for solving relatively small sparse systems, namely the linear code and interpretive list results of the first and second data columns. These procedures are discussed briefly on page 5 . The observer will note the extreme speed and large memory requirements of the former, and the lack of advantage in either speed or memory of the latter. Other comparisons are given in [2][11].

For the partitioned solution, the fractions of the total solution time devoted to computation, I/O, and other partition-related overhead are critical to the evaluation of the algorithm and its implememtation. Table A4 displays these timings for a number of sizes of available local store (S_k).

III. Other Sparse Matrices

Three other matrices were chosen for study (Table A5), ranging from a highly sparse but large power system problem to a finite element problem of large size and greater density than the family cited in Table A2. Credits and references are as follows:

1) Electrical power system, from Mr. Walter Snyder, American Electric Power;

2) Three-dimensional 44-body mechanism model of Boeing 747 landing system [21], from Mr. Keith Brewer of the Flight Dynamics Laboratory of Wright Patterson Air Force Base;

3) Linear, 2 variable, 2 dimensional finite element model of MESFET transistor [22], from Dr. John Barnes, American Microsystems.

Property	Asymptotic Formula	Formula Evaluation for n=5	Experimental value	Error (per cent)
size	2^{2n}	1024	961	+6.2
numeric storage of A (8-byte words)	$72(2^n)$	73728	66248	+11
symbolic storage of A (2-byte words)	$12(2^n)$	12288.	11284	+8.9
average vector length of A	3.	3.	2.93	+.02
numeric storage of LU (8-byte words)	$(124n-376)2^{2n}$	249856	274968	-9.2
symbolic storage of LU (2-byte words)	$32(2^{2n})$	32768	31712	+3.0
column density of LU	15.5n-47.	30.5	35.7	-14.
number of vectors of L	$4.17(2^{2n})$	4226	4327	-2.4
average vector length of L	1.86n-5.64	3.66	4.85	-19.
average vector length of arithmetic operations in factorization	$.882(2^n)/(n-3.35)$	17.7	10.6	+66.

Table A1. Quality of asymptotic formulas for finite element family

Matrix Dimension	Matrix Storage			Inner Loop Computation*	
	A	L	U	Factorization	F. & B. Sub.
9	2.88	1.91	2.62	1.97 (2.31)	2.25 (2.62)
49	2.71	2.74	2.71	2.92 (4.15)	2.53 (3.78)
225	2.87	3.71	2.97	4.42 (6.69)	3.34 (5.29)
961	2.93	4.83	3.86	6.85(10.6)	3.86 (7.07)

*Presented as: successive multiply-subtract(separate multiply-subtract)

(a) Average Vector Lengths (words)

Table A2. Structural properties of finite element experimental problems

Matrix Dimension	LU Storage* (bytes)		Column Density		% non-zero*	
	Numerical (*8)	Symbolic (*2)	A	LU	A	LU
9	336.	62.	5.44	2.33	60.5	58.3
49	4,880.	752.	7.36	6.22	15.0	25.9
225	40,848.	5,376.	8.21	11.3	3.65	10.1
961	267,280.	31,712.	8.61	17.4	.897	3.62

*Unit diagonal not included

(b) Density

Table A2. Structural properties of finite element experimental problems

Matrix Size	Measured Quantity	Code* Generation	Interpreter*	Scalar* LU Map	Vector*+ LU Map	Single Partition** Vector LU Map
9	Symbolic (ms)	6.69	5.67	1.24	5.35	7.3
	Numeric (ms)	.27	.38	.626	(.729),.898,(1.12)	40.9
	F.& B. Sub. (ms)	.22	.52	.247	.339, (.573)	3.2
	Memory (bytes)	2044.	1236.	592.	608.	
49	Symbolic	136.	111.	16.4	20.6	25.3
	Numeric	3.37	14.1	8.8	(8.8),11.4,(14.2)	91.4
	F.& B. Sub.	1.08	3.95	2.1	2.6, (3.6)	7.8
	Memory	36188.	17820.	5072.	4436.	
225	Symbolic	2771.	2374.	197.	131.	138.
	Numeric		203.	108.	(95.),136.,(177.1)	390.
	F.& B. Sub.		29.7	15.	19., (25.)	29.
	Memory	448828.	197788.	33232.	24036.	
961	Symbolic			2039.	792.	813.
	Numeric			1146.	(922.),1309.,(1773.)	2497.
	F.& B. Sub.			94.	113., (156.)	162.
	Memory			189840.	112612.	

{*, **} Unordered matrix symbolic and numeric data assumed in {local, backing} store at beginning of numeric factorization

\+ Entries presented as (time for assembler version), time with combined multiply-subtract inner loop, (time with separate multiply-subtract)

Table A3. Experimental results comparing five sparse matrix solution methods applied to dissected finite element grid; times in milliseconds, memory in bytes; Amdahl 470 V/6, Fortran H, double precision

Maximum Storage (bytes)			T_s	T_p	T_w	T_F(ms)**				T_{FBS}(ms)			
Partition ($2S_K$)	Total*	No. of Partitions	ms	ms	ms	Numeric	I/O	Overhead	Total	Numeric	I/O	Overhead	Total
988K	1048K	1	730.	29.	53.	1667.	318.	512.	2497.	102.	59.	2.	163.
464K	524K	2	730.	91.	53.	1631.	352.	451.	2434.	102.	56.	2.	160.
202K	262K	3	730.	191.	54.	1592.	369.	451.	2412.	103.	55.	2.	160.
71K	131K	9	730	223.	54.	1563.	513.	437.	2513.	104.	59.	2.	165.
13K	73K	47	730.	144.	60.	1714.	1384.	569.	3667.	107.	84.	2.	191.

*Includes 1K driver program, 30.3K constant array storage, and 28.3K numeric solution subroutines

**For blocks F,G,H, Figure 18; I/O times are calculated from:
Read time = .067 + .00023 * NB ms
Write time = .085 + .000217 * NB ms
where NB is the number of bytes transferred.

Table A4. Results of partitioned program on 961 equation problem; Amdahl 470 V/6, Fortran H, double precision.

	Problem Description	No. of Equations	LU Storage Numerical (*8)	LU Storage Symbolic (*2)	Column Density of LU
1.	Electrical Power System	5300	226,448	72,462	5.35
2.	5-Strut 3-D Aircraft Landing System	924	131,240	23,910	17.7
3.	Linear Finite Element Electronic Device Model (Dissected triangular grid)	1688	971,000	77,466	72.1

		L_{ave}, Inner Loop Computation*		Computation Time (ms)	
		Factorization	F. & B. Sub.	Symbolic (S)	Numerical (N)
1.	See above	2.6 (6.1)	1.47 (2.32)	1636.	1105.
2.	See above	1.82 (6.3)	1.62 (2.84)	679.	339.
3.	See above	16.3 (38.2)	8.5 (24.8)	3979.	18400.

*Presented as: successive multiply-subtract (separate multiply-subtract)

Table A5. Results of vectorized solutions of engineering-related sparse matrices (Amdahl 470 V/6, Fortran H)

Appendix B. Example Fortran codes for forward substitution, with separate and combined multiply-subtract

```
C
C  FORWARD SUBSTITUTION
C
      DO 30 I = 1,NM1
         ITOP = IL(I)
         IBOT = IL(I+1)-1
         IVLI = IVL(I)-1
         AUX  = X(I)
         LEN  = IVL(I+1)-IVLI-1
         DO 5 K = 1,LEN
    5       T(K)= AUX*L(IVLI+K)
         IVTI = 0
         IF (ITOP.GT.IBOT) GO TO 30
         J = ITOP
   10        JLJL = JL(J)
             IF (JLJL.LT.0) GO TO 20
             J = J+1
             JLJ = JL(J)
             JDIFF = IVTI - JLJL+1
             DO 15 K = JLJL,JLJ
   15           X(K) = X(K) - T(JDIFF+K)
             IVTI = JDIFF + JLJ
             GO TO 25
   20        IVTI = IVTI + 1
             X(-JLJL) = X(-JLJL) - T(IVTI)
   25        J = J+1
             IF (J.LE.IBOT) GO TO 10
   30    CONTINUE
```

Separate multiply-subtract

```
C
C  FORWARD SUBSTITUTION
C
      DO 30 I = 1,NM1
         ITOP = IL(I)
         IBOT = IL(I+1)-1
         IVLI = IVL(I)-1
         AUX  = X(I)
         IF (ITOP.GT.IBOT) GO TO 30
         J = ITOP
   10        JLJL = JL(J)
             IF (JLJL.LT.0) GO TO 20
             J = J+1
             JLJ = JL(J)
             JDIFF = IVLI - JLJL + 1
             DO 15 K = JLJL,JLJ
   15           X(K) = X(K) - AUX*L(JDIFF+K)
             IVLI = JDIFF + JLJ
             GO TO 25
   20        IVLI = IVLI + 1
             X(-JLJL) = X(-JLJL) - AUX*L(IVLI)
   25        J = J+1
             IF (J.LE.IBOT) GO TO 10
   30    CONTINUE
```

Combined multiply-subtract

Appendix C. Main Programs for Solution of Finite Element Grid of Figure 19

```
      IMPLICIT REAL*8(A-H,O-Z)
C
C   SYMBOLIC MAIN PROGRAM FOR EXAMPLE OF FINITE ELEMENT
C   PREPROCESSING.  SEE VARIABLE DESCRIPTIONS INTERNAL TO
C   SUBROUTINES.
C
C DIMENSIONS FOR ROUTINE UNBLOK
C
      INTEGER*2 IPART,LEN,IA,NA
      DIMENSION IPART(6),IXP(6)
      DIMENSION NA(104)
      DIMENSION MAP(1000),LEN(1000),IPB(1000),JPB(1000)
      DIMENSION JP1(47),IVA(47),JA(47),IA(260)
      INTEGER N/46/
      INTEGER IVEC/0/,NUMBUF/2/,NPART/5/,NBC/8/
      DATA IPART/1,11,21,31,41,47/
      DATA IXP/1,25,49,73,97,97/
      DATA NA/1,2,3,3,4,5,1,2,6,2,3,8,3,4,8,4,5,10,
     C 2,6,8,4,8,10,
     C       6,7,8,8,9,10,6,7,11,7,8,13,8,9,13,9,10,15,
     C 7,11,13,9,13,15,
     C       11,12,13,13,14,15,11,12,16,12,13,18,13,14,18,
     C 14,15,20,12,16,18,14,18,20,
     C       16,17,18,18,19,20,16,17,21,17,18,22,18,19,22,
     C 19,20,23,17,21,22,19,22,23,
     C       1,2,5,9,41,43,45,46/
      INTEGER IMUNIT/0/
C COMMON TO ROUTINE UNBLOK
      COMMON /FEL/ NVAR,NNODE
C COMMON TO ZLIB I/O ROUTINES
      COMMON /ZLEN/ MAXLEN
C
C DIMENSIONS FOR ZMSP, ZBREAK, ZMSPO
C
      INTEGER*2 IX,IXT,IXB,IP,ICNT,IPC,IPRI,KA,IXBUFF
      DIMENSION IBREAK(51)
      DIMENSION IPC(46),IPRI(46)
      DIMENSION ISYMPL(46),ISYMPU(46),IKVA(46),IUPOS(46)
      DIMENSION JU(47),JL(47),IVU(47),IVL(47),LA(47)
      DIMENSION IPTR(47),IXT(47),IXB(47),IP(47)
      DIMENSION KA(260),IX(300)
      DIMENSION JXBUFF(500),IXBUFF(1000)
      EQUIVALENCE (IXBUFF,JXBUFF)
      DATA IPC/1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
     C 21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,
     C 40,41,42,43,44,45,46/
      DATA IPRI/1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
     C 21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
     C 41,42,43,44,45,46/
      INTEGER IAUNIT/1/,INUNIT/2/
      INTEGER MAXCNT/1/,MAXKA/260/,MAXXS/300/
C COMMON TO ROUTINE ZBREAK
      COMMON /SIZE/ MAXTIB,MAXTIX,MAXTA,MAXTZB
C COMMON TO ROUTINE ZBREAK
      COMMON /INTACT/ IPR
C
C
      NVAR= 2
      NNODE= 3
      MAXLEN= 32758
C
C NUMERIC BUFFER SIZES - NOT APPLICABLE IF
C SYMBOLIC AND NUMERIC RUN SEPARATELY
C
      MAXTIB= 10000
      MAXTIX= 10000
      MAXTA= 10000
      MAXTZB= 10000
C
      IPR= 1
```

```
C
C SYMBOLIC FINITE ELEMENT AND BLOCK PREPROCESSING
C
      CALL UNBLOK(N,NA,IXP,NBC,NPART,IPART,IVEC,NUMBUF,IMUNIT,
     C IPB,JPB,JP1,KT,MAP,LEN,IADIM,IVA,JA,IA)
C
C WRITE A NUMERIC BUFFER DIMENSION
C
      ID= IADIM*NUMBUF
      PRINT 1,ID
    1 FORMAT('0DIMENSION A(',I5,')')
C
C SYMBOLIC MATRIX FACTORIZATION
C
      CALL ZMSP(N,JA,IA,IVA,JU,JL,IX,IVU,IVL,IXT,IXB,IP,
     C ICNT,MAXXS,MAXCNT,IPC,IPRI,LA,KA,MAXKA)
C
C INTERACTIVE MATRIX PARTITIONING
C
      CALL ZBREAK(N,IVA,JA,LA,JU,JL,IX,IVU,IVL,NBRKS,IBREAK,
     C ISYMPL,ISYMPU,IKVA,IUPOS,IPTR,&2)
C
C OUTPUT SYMBOLIC MATRIX PARTITIONS
C
      CALL ZMSPO(N,IVA,IA,JA,LA,KA,JU,IX,IVU,IVL,IPC,IPRI,IBREAK,
     C NBRKS,IXBUFF,JXBUFF,ISYMPL,ISYMPU,IPTR,IAUNIT,INUNIT)
    2 STOP
      END
      IMPLICIT REAL*8(A-H,O-Z)
C
C   NUMERIC MAIN PROGRAM FOR EXAMPLE OF FINITE ELEMENT
C   PREPROCESSING.  SEE VARIABLE DESCRIPTIONS INTERNAL TO
C   SUBROUTINES.
C
C DIMENSIONS FOR ROUTINE ABLOK
C
      INTEGER*2 IPART,LEN,NB
      DIMENSION IPART(6),MAP(300),NB(16)
      DIMENSION IVA(47)
      DIMENSION A(248),B(46)
      INTEGER N/46/
      INTEGER IMUNIT/0/,NAUNIT/3/,IBUNIT/4/,NBUNIT/5/
C
C DIMENSIONS FOR ROUTINES ZMNP, ZMBPR
C
      INTEGER*2 IPC,IPRI,IXBUFF,IBBUFF
      DIMENSION IPC(46),IPRI(46),IPTR(46)
      DIMENSION DI(46),X(46)
      DIMENSION XBUFF(308),JXBUFF(616),IXBUFF(1232)
      DIMENSION XBUFFI(223),JBBUFF(446),IBBUFF(892)
      EQUIVALENCE (IXBUFF,JXBUFF,XBUFF,A)
      EQUIVALENCE (IBBUFF,JBBUFF,XBUFFI,IXBUFF(73))
      INTEGER IAUNIT/1/,INUNIT/2/
      INTEGER JAUNIT/8/,ILUNIT/9/,IUUNIT/10/
C COMMON TO ZLIB I/O ROUTINES
      COMMON /ZLEN/ MAXLEN
C
C
      MAXLEN= 32758
C
C NUMERIC FORMULATION
C
      CALL NFINI(N,A,NBUNIT)
C
C NUMERIC FINITE ELEMENT AND BLOCK PREPROCESSING
C
      CALL ABLOK(N,NPART,IPART,NB,MAP,LEN,IVA,A,B,
     C IMUNIT,NBUNIT,NAUNIT,IBUNIT)
C
C NUMERIC MATRIX FACTORIZATION
C
      CALL ZMNP(N,A,IBBUFF,JBBUFF,XBUFFI,IXBUFF,JXBUFF,XBUFF,IPC,
     C IPRI,DI,IPTR,IAUNIT,NAUNIT,IBUNIT,JAUNIT,INUNIT,ILUNIT,
     C IUUNIT)
C
C ROW ORDER FORWARD AND BACK SUBSTITUTION
C
      CALL ZMBPR(N,IXBUFF,JXBUFF,XBUFF,IBUNIT,ILUNIT,IUUNIT,
     C DI,B,X,IPC,IPRI)
C
C WRITE SOLUTION VECTOR
C
      PRINT 1,(B(I),I=1,N)
    1 FORMAT('0SOLN VEC'/(5G14.5))
      STOP
      END
```

REFERENCES

[1.] George, J.A., "Block Eliminations on Finite Element Systems of Equations," Sparse Matrices and Their Applications, Ed. D.J. Rose and R.A. Willoughby, Plenum Press, 1972.

[2.] Woo, P.T., et al, "Application of Sparse Matrix Techniques in Reservoir Simulation," SPE 4544, 48th Annual Fall Meeting of the Soc. of Pet. Engrs., Las Vegas, 1973.

[3.] Bank, R.E., "Marching Algorithms and Block Gaussian Elimination," Sparse Matrix Computations, Ed. by Brunch and Rose, Academic Press, 1975.

[4.] George, J.A., "An Efficient Band-Oriented Scheme for Solving n by n Grid Problems," Proc. AFIPS Conf., FJCC, vol. 41, 1972, pp. 1317-1320.

[5.] IBM System/360 and System 370 Subroutine Library-Mathematics (SL-MATH), no. 5736-XM7 1971 (Rental software available from IBM).

[6.] Calahan, D.A., and T.E. Grapes, "Description of a Sparse Matrix Compiler with Applications," Report AFOSR-TR-71-2676, Systems Engineering Laboratory, University of Michigan, 1971. Also Report AFOSR-TR-72-1973, "Addendum to Sparse Matrix Compiler Manual," Calahan and Schlansker, 1972.

[7.] Tinney, W.F., and J.W. Walker, "Direct Solutions of Sparse Network Equations by Optimally-Ordered Triangular Factorization," Proc. IEEE, vol. 55, pp. 1801-1809, 1967.

[8.] Gustavson, F.G., et al, "Symbolic Generation of an Optimal Crout Algorithm for Sparse System of Linear Equations," Sparse Matrix Proceedings, Ed. by Willoughby, IBM Thomas Watson Res. Center Rpt. RA1 3-12-69, pp. 1-9, 1968.

[9.] Lee, H., "An Implementation of Gaussian Elimination for Sparse Systems of Linear Equations," Sparse Matrix Proceedings, Ed. by Willoughby, IBM Thomas Watson Res. Center Rpt. RA1, 3-12-69, pp. 75-84.

[10.] Gustavson, F.G., "Some Basic Techniques for Solving Sparse Systems of Linear Equations," Sparse Matrices and Their Applications. Ed. by Rose and Willoughby, pp. 41-52, 1972.

[11.] Woo, P.T., et al, "Application of Sparse Matrix Techniques to Reservoir Simulation," Sparse Matrix Computations, Ed. by Bunch and Rose, Academic Press, 1976.

[12.] Chang, A., "Application of Sparse Matrix Methods in Electric-Power System Analysis," Sparse Matrix Proceedings, Ed. by Willoughby, IBM Thomas Watson Res. Center Rpt. RA1 3-12-69, pp. 113-122, 1968.

[13.] Calahan, D.A., "Parallel Solution of Sparse Simultaneous Linear Equations," Proc. Eleventh Allerton Conference on Circuit and System Theory, University of Illinois, pp. 729-738, 1973.

[14.] Von Fuchs, G., J.R. Roy, and E. Schrem, "Hypermatrix Solution of Large Sets of Symmetric Positive - Definite Linear Equations," Computer Methods Appl. Mech. Enging., vol. 1, pp. 197-216, 1972.

[15.] Noor, A.K., and S.J. Voigt, "Hypermatrix Scheme for Finite Element Systems on CDC STAR-100 Computer, Computers and Structures, vol. 5, pp. 287-296, 1975.

[16.] Gentleman, W.L., and A. George, "Sparse Matrix Software," in Sparse Matrix Computations, Ed. by Bunch and Rose, pp. 243-262, Academic Press, 1976.

[17.] McKellan, A.C., and E.G. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems," CACM, vol. 12, no. 3, March, 1969.

[18.] Eisenstat, S.G., et al, "Considerations in the Design of Software for Sparse Gaussian Elimination," Report #55, Dept. of Computer Science, Yale University, (no date)

[19.] Knight, J.C., W.G. Poole, and R.G. Voigt, "Systems Balance Analysis for Vector Computers," ICASE Report 75-6, NASA Langley Research Center, March 1975.

[20.] Hachtel, Gary, "The Sparse Tableau Approach to Finite Element Assembly," Sparse Matrix Computations, Ed. by Bunch and Rose, Academic Press, 1976.

[21.] Orlandea, N., D.A. Calahan, and M.A. Chace, "A Sparsity-Oriented Approach to the Dynamic Analysis and Design of Mechanical Systems, Parts I and II," Journal of Engineering for Industry, (to be published).

[22.] Lomax, R.J., and J. Barnes, "Two-dimensional Finite Element Simulation of Semiconductor Devices," Electronics Letters, vol. 10, no. 16, pp. 341-3, 8 August, 1974.